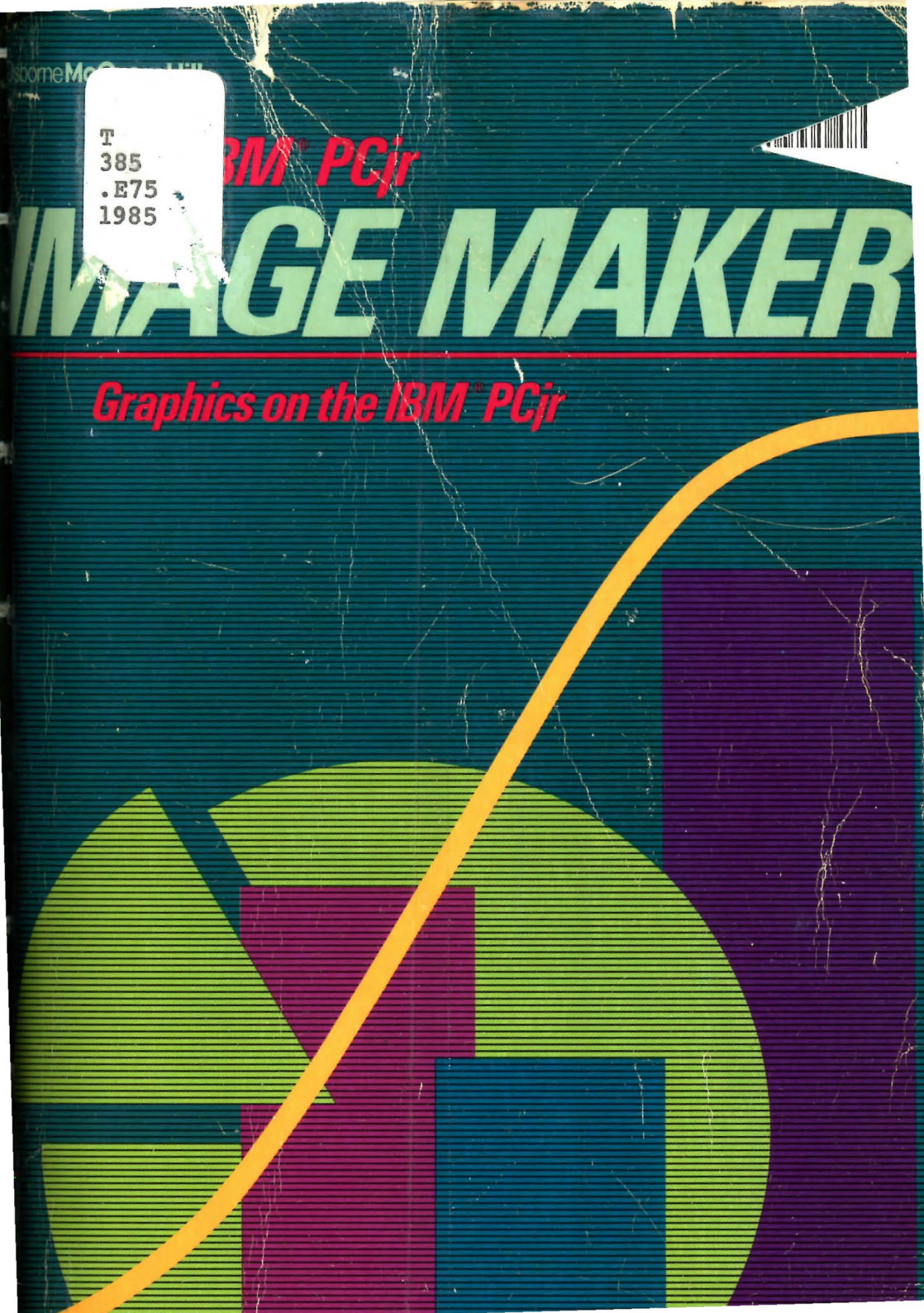
T
385
.E75
1985

IBM® PCjr IMAGE MAKER

Graphics on the IBM® PCjr



THE IBM® PCjr

IMAGE MAKER:

GRAPHICS ON THE IBM PCjr

Jonathan Erickson
William D. Cramer

Osborne **McGraw-Hill**
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book
distributors outside of the U.S.A., please write to
Osborne**McGraw-Hill** at the above address.

IBM is a registered trademark of International Business Machines.

**THE IBM® PCjr IMAGE MAKER:
GRAPHICS ON THE IBM PCjr**

Copyright ©1985 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DODO 8987654

ISBN 0-88134-138-X

Karen Hanson, Acquisitions Editor
Denise Penrose, Technical Editor
Paul Hoffman, Technical Reviewer
Carol Hamilton, Copy Editor
Richard Cash, Book Design
Bonnie Bozorg, Composition
Yashi Okita, Cover Design

Acknowledgments

A special thanks to Doug Kneibert and Clariece Washlick.

J.E.

TABLE OF CONTENTS

Introduction	ix	
Part 1	**Graphics and Your PCjr**	1
Chapter 1	Graphics Overview	3
Chapter 2	Graphics and the PCjr	7
Part 2	**Getting Started With PCjr Graphics**	21
Chapter 3	Initializing the Screen: SCREEN and CLS	23
Chapter 4	Controlling Color: COLOR, PALETTE, and PALETTE USING	37
Chapter 5	Turning Pixels On and Off: PSET and PRESET	55
Chapter 6	Creating Lines and Rectangles: LINE	69
Chapter 7	Drawing a Circle, Semi-Circle, or Arc: CIRCLE	89
Chapter 8	Drawing "Free-Hand": DRAW	111
Chapter 9	Filling Areas with Color: PAINT	135
Chapter 10	Simulating Movement: GET and PUT	151
Chapter 11	Moving Graphics Between Pages: PCOPY	167
Chapter 12	Redefining the Screen: VIEW and WINDOW	171
Chapter 13	Identifying Screen Definitions: POINT and PMAP	189

Part 3	**PCjr Graphics Applications**	195
Chapter 14	Creating Bar Charts and Graphs	197
	Photo Section	215
Chapter 15	Creating Line Charts and Graphs	223
Chapter 16	Creating Pie Charts and Graphs	233
Chapter 17	Creating Function Plots	251
Chapter 18	Designing and Displaying Special Characters	261
Chapter 19	Icons and Menus	283
Chapter 20	Animation	291
Part 4	**Appendixes**	297
Appendix A	Binary/Decimal/Hexadecimal Base	299
Appendix B	Graphics Worksheet	305
Index		309

INTRODUCTION

The PCjr Image Maker

Vivid, full-color computer graphics is one of the most exciting applications of the IBM PCjr. *The PCjr Image Maker* provides you with the necessary tools to understand and use the powerful graphics capabilities of your PCjr. We start by describing each graphics statement in detail. Then we show you how to use those commands to create everything from business graphs to video games.

The PCjr Image Maker is divided into four sections. Part 1 gives you an overview of graphics: what graphics is in general and computer graphics in particular. Part 2 explains individual graphics commands, starting with the statements you will use first in your programs. Each explanation includes sample programs that illustrate the concept under discussion. Part 3 shows you how to apply the commands introduced in the previous section. Complicated graphics applications (animation, icon creation, and others) are presented and explained so that you can easily adapt any of the routines in programs you write. Part 3 also contains a color photo section of actual screen displays produced by the programs. Although the photo section follows Chapter 15, it contains photos from Chapter 14 on. Finally, Part 4 provides a number base conversion chart and a graphics worksheet to plot your images before you create them.

What this book won't do is teach you all there is to know about programming the PCjr. If you have not already done so, you should read *Hands-On BASIC* and become familiar with programming concepts (loops, arrays, and so forth) before you begin to read *The PCjr*

Part 3 **PCjr Graphics Applications** 195

Chapter 14 Creating Bar Charts and Graphs 197

Photo Section 215

Chapter 15 Creating Line Charts and Graphs 223

Chapter 16 Creating Pie Charts and Graphs 233

Chapter 17 Creating Function Plots 251

Chapter 18 Designing and Displaying Special Characters 261

Chapter 19 Icons and Menus 283

Chapter 20 Animation 291

Part 4 **Appendixes** 297

Appendix A Binary/Decimal/Hexadecimal Base 299

Appendix B Graphics Worksheet 305

Index 309

INTRODUCTION

The PCjr Image Maker

Vivid, full-color computer graphics is one of the most exciting applications of the IBM PCjr. *The PCjr Image Maker* provides you with the necessary tools to understand and use the powerful graphics capabilities of your PCjr. We start by describing each graphics statement in detail. Then we show you how to use those commands to create everything from business graphs to video games.

The PCjr Image Maker is divided into four sections. Part 1 gives you an overview of graphics: what graphics is in general and computer graphics in particular. Part 2 explains individual graphics commands, starting with the statements you will use first in your programs. Each explanation includes sample programs that illustrate the concept under discussion. Part 3 shows you how to apply the commands introduced in the previous section. Complicated graphics applications (animation, icon creation, and others) are presented and explained so that you can easily adapt any of the routines in programs you write. Part 3 also contains a color photo section of actual screen displays produced by the programs. Although the photo section follows Chapter 15, it contains photos from Chapter 14 on. Finally, Part 4 provides a number base conversion chart and a graphics worksheet to plot your images before you create them.

What this book won't do is teach you all there is to know about programming the PCjr. If you have not already done so, you should read *Hands-On BASIC* and become familiar with programming concepts (loops, arrays, and so forth) before you begin to read *The PCjr Image Maker.* If you are familiar with *Hands-On BASIC,* you know

that it discusses graphics in a very perfunctory manner. We begin where it left off.

To use the programs described in this book, you must have an enhanced PCjr system—Cartridge BASIC, at least 128K of RAM, and a minimum of one disk drive. Furthermore, all programs require that you use BASICA, not BASIC. This means that you must type **BASICA** and press ENTER when the DOS A> prompt is displayed.

Each chapter in Part 2 of *The PCjr Image Maker* focuses on one or two graphics commands. Individual commands are summarized in a special *syntax block* introduced at the beginning of each command description. This summary includes the command statements that *must* be typed in as well as all of the options and parameters associated with the command. **BOLDFACE** words in all capital letters and punctuation must be typed in *exactly* as shown. Words in *lowercase italics* are variables for which you must supply a value.

P A R T O N E

Graphics And Your PCjr

*Part I provides you with the background information you need to get started with computer graphics. Many important terms and concepts used throughout the book (*pixel, coordinates, *and so on) are introduced and defined in this section. If you are already familiar with computer graphics to some degree, reading this section will make you aware of some of the unique graphics features and capabilities of the PCjr.*

CHAPTER ONE

Graphics Overview

Drawings, photographs, charts, and other forms of pictorial representation are collectively referred to as "graphics." The term graphics has evolved to describe not only the pictures, but also the technologies that produce them. Today, graphics includes cartoons, video games, business graphs, architectural drafting, mechanical design, greeting cards, printing, typesetting, photography, and a host of other technologies and applications.

Most people grasp ideas and concepts that are presented visually much faster than those read in text. In fact, if you think back to the storybooks of your childhood, the pictures probably told you more than the words. People everywhere respond readily to all forms of graphics because pictures possess universal significance and therefore cross linguistic barriers. A picture of a house means the same to a speaker of French as to a speaker of English.

Throughout history, artists and scientists have applied the available technologies and mediums to produce graphics. Plant dyes were used to paint cave walls, paper and ink for hand-lettered and illustrated books and maps. Later, artists used woodcuts, lead type, silk screens, and eventually photography to transfer images and convey ideas.

Today, computers are being used to make the entire process of graphics design and production easier and more expressive. Everything from calendars to skyscrapers is being designed with the help of computers. (This is referred to as CAD—"computer-aided design.") Computer graphics is not intended to replace artists and their materials, but

to free graphics artists from mechanical tasks, allowing them to concentrate on creative expression. Computers also provide artists with a number of different media—the most common being a television screen and the paper in a printer.

WHAT IS COMPUTER GRAPHICS?

Technically speaking, "graphics" uses lines to represent a three-dimensional object on a two-dimensional surface. In most instances, these lines are laid out according to standard mathematical rules; that's why graph paper is drawn with mathematically precise lines and distances. A line can be any length—from a dash that is so short you can't see it, to a line that continues into infinity. Furthermore, a line doesn't have to be straight; it can be an arc or a curve. Figure 1-1 provides enough information for you to visualize a three-dimensional object, even though the drawing is rendered on a flat surface.

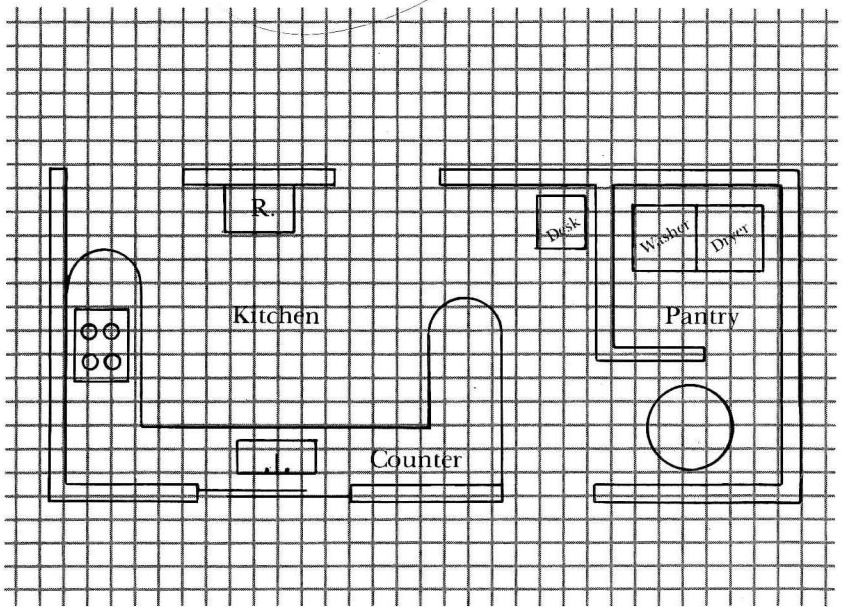


Figure 1-1. *Partial house plan*

With computer graphics like those generated on your PCjr, the two-dimensional surface is usually a television screen. In other cases, it may be paper on which your IBM graphics printer prints. Typically, computer-generated graphics includes graphs, illustrations, symbols, cartoons, animation, and other forms of artwork. Video games found in arcades and weather maps shown on the 6 o'clock news require machines far more powerful than the PCjr.

For computers like your PCjr, all of the lines that create artwork are based on a single dot or picture cell on the screen called a *pixel.* A line is simply a row of pixels. When the pixels are very close together, they appear to be a solid line. (You may have heard of half-tone photography or of the "pointillism" school of painting. Both techniques are similiar to pixel graphics in that they depend on thousands of dots to create an image.) Other types of computers use different forms of technology to create a line, but for low cost and maximum flexibility, pixel graphics is the best.

Besides having the capability to draw lines of varying lengths and directions, computers like your PCjr can display colors. Of course, to see those colors, you will need a color monitor or color television attached to your computer. If you have a color printer attached to your computer, you can also make printouts. Most personal computers can display 4 to 16 colors; your IBM PCjr provides 16. You can still give the appearance of additional colors by using a shading technique called "tiling," which is discussed later in this book.

If you have ever played a video game, you know that computer graphics systems also allow you to simulate motion or animation. Different computers accomplish this differently. Your PCjr lets you simulate motion in two ways. One way is by drawing an object and moving it around the screen. Another is by drawing objects at slightly different locations on different "pages," then flipping or displaying the pages very quickly. Of course, all of this requires a great deal of speed, but that's another thing computers are good at—getting things done in a hurry.

As you become familiar with your PCjr's graphics capabilities, your computer will provide more fun and challenges than you ever expected.

CHAPTER TWO

Graphics And the PCjr

There are two elements you must consider when displaying graphics: color (both foreground and background) and image resolution. These display capabilities are interrelated; the number of available colors depends on the resolution desired—and vice versa. Examining each of the capabilities and understanding how they interrelate is the first step toward writing a graphics program.

The PCjr can display two types of information on the screen: text (letters and numbers) and graphics (circles and lines). To display these types of information, the computer must first be set to the appropriate method of display, called a *mode.* The PCjr has two display modes:

- *Text Mode* displays numbers and letters only.
- *Graphics Mode* displays circles, lines, and other graphics forms, in addition to numbers and letters.

When you first turn on your PCjr's power and load BASICA, the computer is automatically set to display text only; it won't display graphics even if you try. To display graphics information (or to change from Graphics Mode back to Text Mode), you must switch from one mode to the other using the SCREEN statement. This is sometimes called a *software switch,* since it is similar to turning a switch like a television's channel selector. In this case, however, you don't physically turn a knob; you just type in the SCREEN statement and the computer "turns" the switch for you, changing the display mode.

TEXT MODE

For most computer operations, Text Mode may be considered the PCjr's normal mode of operation. It is the mode in which you perform traditional "number crunching" computing tasks. For instance, if you want to write or use a program that computes amortization schedules, you would certainly want to use Text Mode. Because Text Mode provides you with a larger memory area than Graphics Mode does, you can write longer, more powerful programs. (Graphics Mode, on the other hand, uses a part of *your* memory area just to execute its graphics routines.)

You can type programs in either Text or Graphics Mode. The results of programs written in Text Mode can be displayed in either mode; the results of graphics programs can be displayed only in Graphics Mode. For instance, you can be in Text Mode when you *write* a program that draws a circle, but you must be in Graphics Mode when the circle is actually drawn on the screen.

Text Mode gives you the option of either 40 or 80 characters (columns) displayed across the screen. (The WIDTH command allows you to select one or the other.) Both Text Mode column selections have 25 lines (or rows) down the screen. See Figure 2-1.

Both 40- and 80-column modes allow you to access all 16 colors available on the PCjr. There are several reasons why you might want to control the color in Text Mode. You might find one color easier on your eyes than another. You might use a special color to highlight or emphasize something on the screen, such as a particular paragraph on a page of text. If you've used any application programs (such as word processing) that are specifically designed for the PCjr, you'll notice that color has often been used to make the program easier to operate. To control the colors, use the COLOR statement. This statement also controls colors in Graphics Mode, but is used somewhat differently.

GRAPHICS MODE

While Text Mode has two display options (40- and 80-column), Graphics Mode allows six display options. You use the SCREEN command to specify which of the six choices you want.

There are two obvious differences between the choices: the *resolution*, or number of pixels in a specific area of the screen, and the number of colors immediately available.

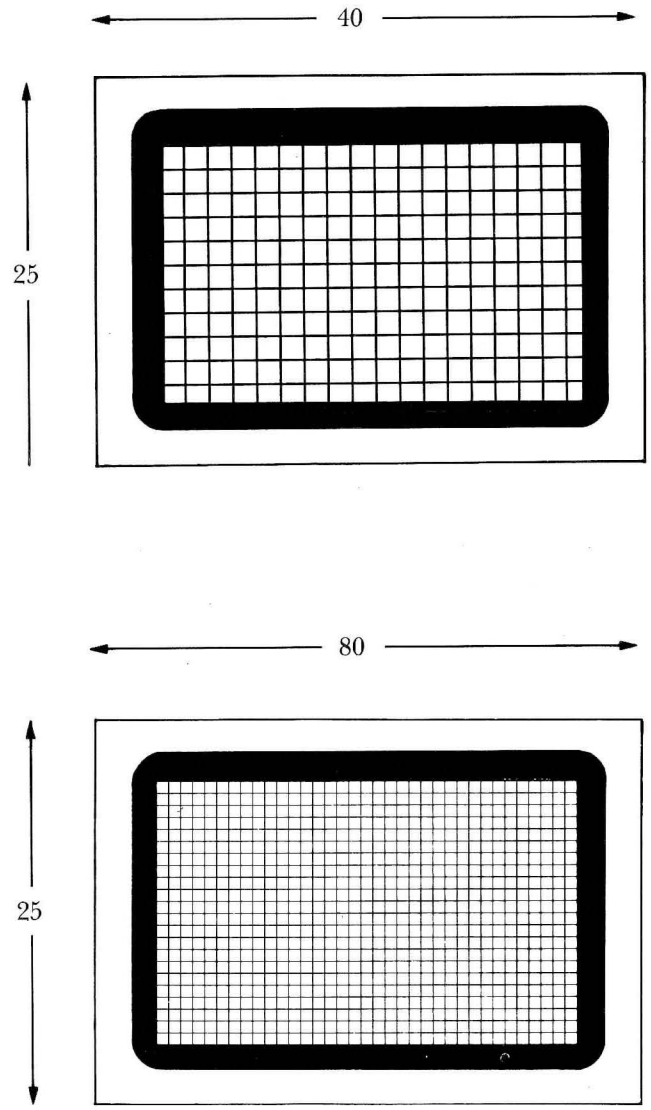


Figure 2-1. *The PCjr's 40-column and 80-column Text Mode display*

Resolution is important because the greater the number of pixels, the sharper the image. The PCjr has three levels of resolution, each with a different number of pixels:

- Low resolution has 160 pixels across the screen and 200 up and down (160 × 200 = 32,000 pixels).
- Medium resolution has 320 pixels across the screen and 200 up and down (320 × 200 = 64,000 pixels).
- High resolution has 640 pixels across the screen and also 200 up and down (640 × 200 = 128,000 pixels).

High resolution makes more pixels available on the screen at any one time, allowing for finer detail and consequently a sharper image. Low resolution obviously has relatively few pixels on the screen, while medium resolution is the middle ground.

Notice that although we have described only three, there are indeed six graphics display options. These options include one low resolution, three different medium resolutions, and two high resolutions. The different medium resolutions and the high resolutions are distinguished by the available colors and memory in each mode. These concepts are discussed in Chapter 3.

Since the screen size cannot physically change to accommodate the additional dots required by medium and high resolution, the size of the dots must change: the greater the screen resolution, the smaller the pixels. Therefore, both the number and size of the pixels change in medium and high resolution. Figure 2-2 shows the difference in relative

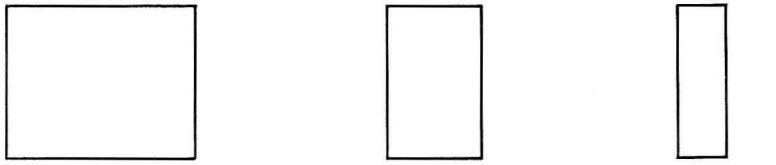


Figure 2-2. *Relative pixel sizes*

size between a high-resolution and a low-resolution pixel. Keep in mind, however, that these aren't shown actual size.

To illustrate the effect of the size difference among the pixels, run the following three programs, one at a time. Don't worry if you must enter commands that are unfamiliar to you; just observe the program results. Notice that in each instance, a pixel will light up in the center of the screen.

```
10 SCREEN 3
20 CLS
30 PRINT "Low Resolution"
40 PSET (60,100)
50 END
```

```
10 SCREEN 1
20 CLS
30 PRINT "Medium Resolution"
40 PSET (160,100)
50 END
```

```
10 SCREEN 2
20 CLS
30 PRINT "High Resolution"
40 PSET (320,100)
50 END
```

The number of readily available colors is another basic difference among the graphics display modes. The PCjr, like most personal computers, does not offer both high resolution and numerous colors. (There are a number of reasons for this, most of which are beyond the scope of this book.) That's why low resolution, for instance, makes 16 colors available without extra programming, while high resolution provides only four colors initially.

X,Y COORDINATES

As we mentioned, the PCjr's highest-resolution mode puts 128,000 pixels on the screen. When you begin writing programs, you must learn the location of all of the pixels for the specified resolution in order to tell the PCjr where to begin and end an image. For example, to draw a

line on the screen, you need to specify the exact pixel at which to begin drawing the line and the exact pixel at which to stop. You can identify any one of those thousands of tiny dots and turn it on or off like a light bulb. Keeping track of each pixel, however, isn't as difficult as you might imagine. To do so, use an X,Y coordinate system just like that on a piece of graph paper.

As you can see in Figure 2-3, X-values are counted horizontally across the grid (in columns) and the Y-values are counted vertically up and down (in rows). When you're identifying points on a grid that represents the PCjr display screen, begin counting both the X- and Y-values in the upper left corner, and count left to right. (If you've used grid mapping before, this may seem backwards. Don't worry, though; this is the way most personal computers use X,Y coordinate systems.) Remember that in all resolutions, you begin counting at 0, not 1. In medium resolution, this means that each line has 320 pixels across, the first number on the first line is 0, and the last number on

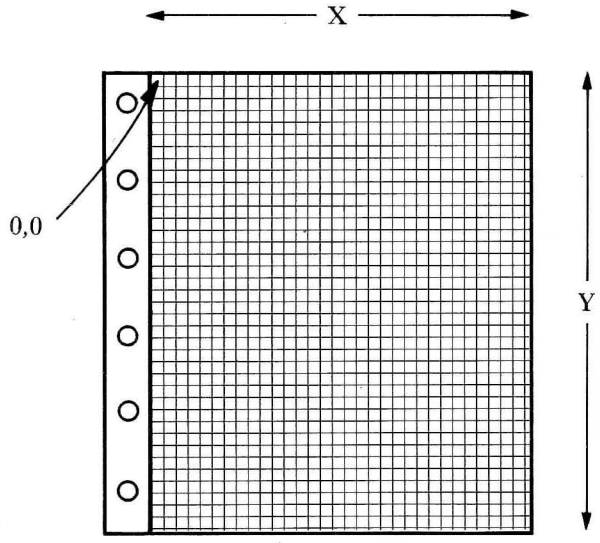


Figure 2-3. *Grid paper and X,Y directions*

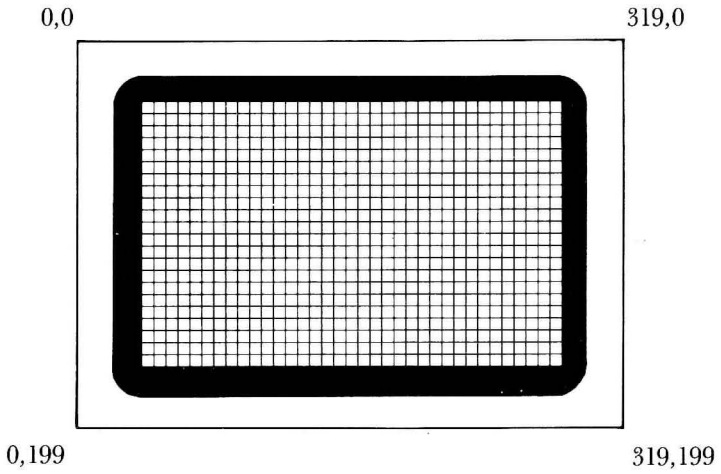


Figure 2-4. *Medium-resolution graphics screen with numbering system*

the first line is 319. In high resolution, you will count across from 0 to 639. See Figure 2-4.

Remember that to specify a point for the drawing to begin, you must indicate a pixel in X,Y terms; the X-value must always precede the Y-value. For instance, you might say (in BASIC form), "begin drawing a line at X,Y = 10,3 and stop drawing the line at X,Y = 100,100." The PCjr would count over to X=10, and then from that point count down to Y=3. It would remember that point, go back to the upper left corner and count over to X=100, and then down to Y=100. The computer would then draw a line connecting the two points and wait for your next instruction. In program form, the instructions might look like this:

```
10 SCREEN 1
20 CLS
30 LINE (10,3) - (100,100)
40 END
```

In many cases, the PCjr provides *defaults* for X,Y values if you don't

identify a starting point for a graphics routine. This means that if you don't assign a value, the computer will use a predetermined one. Typically, the PCjr will use one of two default values for screen locations: either the center of the screen or the *last referenced point (LRP)* in the previous graphics statement. Notice how the center of the screen is used as the LRP in the following program:

```
10 SCREEN 1
20 CLS
30 LINE - (100,100)
40 END
```

Another form of X,Y positioning is called *relative* positioning. In this method of pixel identification, all points are specified in terms of their relationship to the previous point (LRP), rather than to an absolute point. This is somewhat like giving directions to a stranger in town. You could, for example, say that City Hall is located at 110 Main Street; this is absolute positioning. Or you could say that to get to City Hall, the stranger must go to the first stop light, turn left, then go three blocks to the big brick building; this is relative positioning.

Even though X,Y counting starts with 0,0 in the upper left corner, the PCjr does provide a high degree of flexibility in its coordinate numbering. You can use the WINDOW and VIEW statements to redefine both the size of the screen and the coordinate system—allowing you, for instance, to specify 0,0 as the lower right of the screen. This is useful for applications described in a later chapter.

USING COLORS WITH PCjr GRAPHICS

The graphics capabilities of the PCjr can be further enhanced with color. You will discover that adding color to your graphics design is easier than creating the actual design—once you've understood a few simple concepts.

Foreground and Background Colors

The real key to learning how to use colors with the PCjr is to grasp the concepts of *foreground* and *background* colors. To start, imagine the background as a blackboard or chalkboard. The foreground is like the

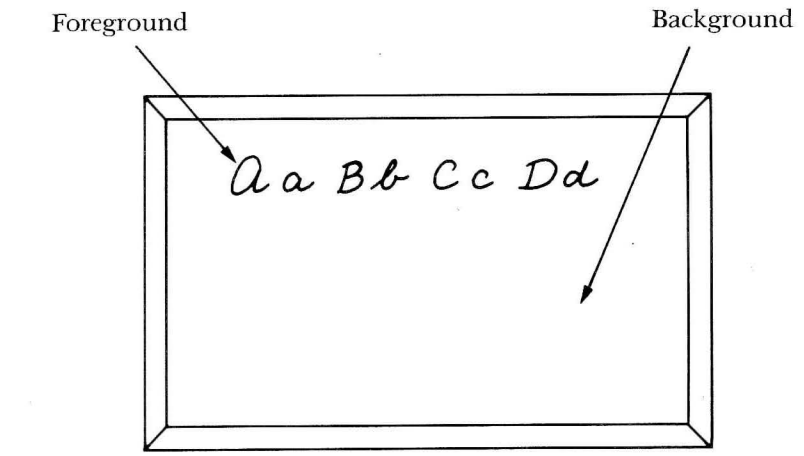


Figure 2-5. *PCjr foreground and background colors*

chalk you use to write on the blackboard. See Figure 2-5.

Obviously, it's a simple matter to write on a blackboard with different-colored chalk. In other words, the foreground color can change. And just as obviously, the background color can change.

Before discussing how you actually control the foreground and background colors on the PCjr, we should consider that each display mode (Text and all three Graphics Modes) has specific color characteristics.

In Text Mode, the characters (letters and numbers) as they appear on the screen are displayed as foreground colors (just like chalk marks on the blackboard). You can control—change—the foreground or background color in both the 40- and 80-column widths. (Remember, however, that if you display a white character on a white background, you won't be able to see it.) If you change the background color, the change will only affect the area immediately around the character, making it appear that the character is sitting inside a colored box.

Graphics Mode presents a different kind of problem. Since there is a trade-off between the degree of resolution and the number of colors, you can't have all 16 colors instantly available in all modes. Typically, the higher the resolution, the fewer the available colors. This is why

Table 2-1. *Total Number of Colors Immediately Available for Each Screen Display Mode*

Screen	Resolution	Number of Colors
0	Text	16
1	Medium	4
2	High	2
3	Low	16
4	Medium	4
5	Medium	16
6	High	4

high-resolution mode has only four available colors. When you write a program for the PCjr, you will have to decide which is more important for that program — higher resolution or more colors — and then choose the appropriate mode.

In medium-resolution Graphics Mode, color becomes a very powerful element. You have the full range of as many as 16 different background colors, plus relatively high resolution.

That's the easy part. It becomes confusing when you also consider foreground colors. Each display mode offers different colors and different ways to access those colors. In low resolution and one of the medium-resolution modes, you have all 16 colors instantly available at any one time for both foreground and background. However, with the 4-color medium-resolution and high-resolution modes, only four colors are instantly available, although the other 12 colors can be accessed with additional programming. With the 2-color high-resolution mode, only two colors are available without additional programming. Table 2-1 shows the total number of colors (foreground and background) available in each screen mode. Notice that the number of colors distinguishes the two high-resolution and medium-resolution modes.

The PCjr has assigned each of the available colors a code number.

When you want to use a certain color, you type in the color's code instead of the name of the color. (Table 2-2 lists colors and color codes.) For instance, to use red, you would include code **4** in your program instead of the word **red**.

In the 16-color Graphics Modes (low resolution and a medium resolution), you specify the code of the color you've chosen. In the 4-color modes (medium and high resolution), however, additional programming is required to access the other 12 colors. You may find it helpful to consider the four available colors as selections on an artist's palette. (Unlike an artist, you cannot mix colors on the PCjr palette; you can only select colors from the palette.)

Table 2-2. *PCjr Colors and Color Codes*

Color Code	Color Displayed
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Gray
8	Light gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	White

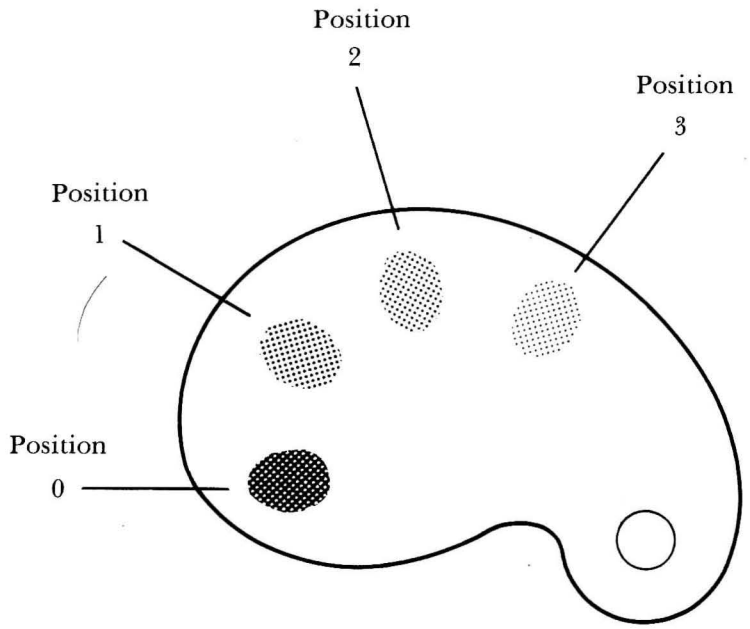


Figure 2-6. *An artist's palette, which is similiar to the PCjr's palette*

As you can see, the palette in Figure 2-6 contains numbered positions for colors, not the colors themselves. In the high-resolution display mode, for instance, the colors black, cyan, magenta, and white have been assigned to positions 0, 1, 2, and 3 respectively. You can assign another color—red, for instance—to one of the positions if you want to use it, but *it will replace an existing color.* To use different colors, you must assign them to specific positions on the palette, and there are only four positions.

Controlling Colors

The basic command used to control color is called COLOR. The COLOR command itself does not always put a color on the screen; instead it tells other commands (PAINT, LINE, and so forth) what you want as the foreground and background colors. Normally, black (code

0) is the background colors while the highest available color code, such as white (code 15), is the foreground color. Handling colors in this manner allows you to change colors dozens of times in a long program without changing every statement that puts colors on the screen; you simply change the COLOR statement.

Another command, PALETTE, allows you to reassign colors to color codes. When you are in 4-color high-resolution mode, for example, and want to use red as the background color, you might assign color code 4 (see Table 2-2) to palette position 0 using PALETTE (see Figure 2-6). Then use COLOR to assign 0 as the background color.

In addition to controlling background and foreground colors, it's also possible to specify the color of objects. Almost every graphics command allows you to specify a color when creating an object. You can, for example, draw a red line if red is available for the display mode you're using. In some cases, you may not specify a color and must simply use the default color; in such cases, the current foreground color will be used.

OVERVIEW OF PCjr GRAPHICS BASIC PROGRAMMING

In general, writing graphics programs is really no different from writing programs for other applications. The programming steps are quite similar.

The first lines of any program fulfill the *initialization* steps. Typically, this includes setting all counters to zero, clearing all buffers of data, and opening any files. Specific procedures for PCjr graphics start with screen initialization, which includes setting aside enough memory, selecting the display modes (low, medium, or high resolution), and clearing the screen. You might also want to reassign colors on the current palette and set the foreground and background colors.

Next are the *processing* steps. These include calculating numbers and displaying or printing results. If you are drawing a pie chart, for instance, you might specify the circle's center point and set the radius, and then draw the object on the screen.

The last step is *finalization*—the step that ends the program. This may include saving displays or data on disk, closing files, and exiting the program.

The following program is a short example of a typical graphics program that incorporates these steps. We won't attempt to explain

what happens in this program (that will come later). Instead, we want you to notice the order of events that goes into writing every graphics program—simple or complex.

```
5  REM Initialization starts here
10 CLEAR,,,32768  'Clears memory for
   high resolution
20 SCREEN 6       'Selects display mode
   for high resolution
30 CLS            'Clears the screen
40 REM Processing starts here
50 CIRCLE (320,100),50  'Draws circle at
   the center of screen
60 REM Finalization starts here
70 END
```

Chapters 3 and 4 describe the initialization and finalization steps. Chapters 5 through 12 describe the processing steps required for writing graphics programs on the PCjr.

P A R T T W O

Getting Started With PCjr Graphics

Understanding PCjr graphics means understanding the different commands that enable you to create images. This section describes each graphics command and function in detail. Part II also provides a summary of each statement for quick reference.

CHAPTER THREE

Initializing the Screen: SCREEN and CLS

Like most procedures, graphics programming requires both forethought and preparation. In this chapter, we'll show you how to begin planning graphics programs and how to prepare your PCjr for them.

The two commands we'll discuss, SCREEN and CLS, really don't put anything on the display at all; instead, they initialize the PCjr screen by preparing it for graphics programs. Together they set up the computer for either text or graphics. If Graphics Mode is selected, the resolution mode and color options are chosen. Other graphics commands create the image, and **CLS** is again used to erase the display. SCREEN, the more complicated of the two statements, performs several other functions that we'll describe in this chapter.

There is one other statement that should be considered when initializing the PCjr screen—CLEAR. Since it's not specifically a graphics command, we won't examine it thoroughly. However, we will discuss how it applies to graphics.

THE SCREEN STATEMENT

It's a good idea to begin a program with a SCREEN statement because it sets up the PCjr for either graphics or text displays. (The one exception to this, the use of the CLEAR statement, is described later in the chapter.) Syntax 3-1 describes the SCREEN statement's parameters.

SCREEN *display mode, color on/off, apage, vpage, erase*

display mode specifies the display mode you want to use. It is a numeric expression from 0 to 6:

0 = Text Mode (40- or 80-column)
1 = Graphics Mode (medium resolution, 320 × 200, 4 colors)
2 = Graphics Mode (high resolution, 640 × 200, 2 colors)
3 = Graphics Mode (low resolution, 160 × 200, 16 colors)
4 = Graphics Mode (medium resolution, 320 × 200, 4 colors)
5 = Graphics Mode (medium resolution, 320 × 200, 16 colors)
6 = Graphics Mode (high resolution, 640 × 200, 4 colors)

color on/off (or *burst*) specifies whether or not you want to use full color. It is a numeric expression of either 0 or non-zero. *Color on/off* does not affect SCREENs 2, 3, 5, or 6; *color on* is always on. In SCREENs 1 and 4, 0 turns color on and a non-zero value turns color off. In SCREEN 0, 0 turns color off and a non-zero number turns it on.

apage specifies the part of video memory in which objects (graphics or text) are created—the "active page." *apage* is a numeric expression between
0 and 48 for display mode 0 (Text Mode)
0 and 6 for display modes 1 through 4
0 and 3 for display modes 5 and 6.
The maximum value for *apage* depends on the current amount of video memory set aside with CLEAR. *apage* is optional; if omitted, page 0 is used.

vpage is the part of video memory you see on the screen—the "visual page"—and is a numeric expression between
0 and 48 for display mode 0 (Text Mode)
0 and 6 for display modes 1 through 4
0 and 3 for display modes 5 and 6
vpage is optional; if omitted, the *vpage* value is used.

Syntax 3-1. *The SCREEN Statement*

erase is the video memory to erase. It is a numeric expression from 0 to 2 and is optional; if omitted, 1 is used.

Syntax 3-1. *The SCREEN Statement (continued)*

Remember that when you first turn on your PCjr's power and enter BASICA, you'll be in Text Mode and you can't use graphics commands. To enter Graphics Mode, you must use the SCREEN command. This doesn't mean you'll only use SCREEN at the beginning of a program. SCREEN also allows you to switch display modes—from text to graphics, for instance—from one Graphics Mode to another, or from one memory "page" to another at any point in a program.

When you first turn on the PCjr's power, SCREEN is automatically set as follows: Text Mode, color off, active page = 0, visual page = 0, and erase = 1. These are called the default values, and the results would be the same as typing in **SCREEN 0,0,0,0,1** and pressing ENTER.

For the most part, you will use the SCREEN statement in a program line. However, you can enter the command in the Direct Mode, and it will remain in effect until you enter different values for SCREEN, turn the PCjr's power off, or encounter an error condition.

Figure 3-1 emphasizes the importance of including at least one operand in a SCREEN statement.

SCREEN 1	SCREEN
SCREEN 2,1	
Legal!	*Illegal!*

Figure 3-1. *One operand must be included in the SCREEN statement or a Missing Operand error will result*

Selecting the Display Mode

SCREEN 0 puts the PCjr in Text Mode. If you are only interested in turning the computer on and writing "number crunching" programs (such as programs that keep track of your utility bills), you shouldn't have to worry about using the SCREEN statement at all; it is automatically set to SCREEN 0 when you turn the PCjr's power on. However, if you want to write graphics programs or programs that make use of other special features (switching from one graphics resolution mode to another, for instance), you must use SCREEN.

If you enter the wrong display mode, an error will occur or the results will not be what you wanted. For example, if you specify that you're in Text Mode (SCREEN 0), you will not be able to execute a graphics routine like drawing a circle. Or if you are in high-resolution Graphics Mode, for instance, and you try to use 16 colors at one time, you will be unsuccessful, since high resolution allows a maximum use of four colors without extra programming. In the following short program, the CIRCLE statement is used to draw a circle. But if you change this program to Text Mode, an error will occur because the display mode is set to SCREEN 0, Text Mode.

```
10 SCREEN 0
20 CIRCLE (100,100),20
30 END
```

Another error condition will result if you use a value that is outside the allowable range. Figure 3-2 shows SCREEN statements that are legal or illegal depending on whether the value chosen is within the allowable range for the display mode.

SCREEN 0	SCREEN 7
SCREEN 1	SCREEN 9
SCREEN 4	SCREEN 15
Legal!	*Illegal!*

Figure 3-2. *Legal and illegal display mode values*

Since the display mode value can be any numeric expression, you can also use variables to represent allowable display modes. The following is perfectly acceptable:

```
10 A = 2
20 SCREEN A
```

But if A = 15, an error would result if you attempted to run the program because 15 is outside the allowable range for display modes.

Selecting SCREENs 5 or 6 and the CLEAR Statement

Important Note! You must set aside 32K of video memory before trying to use SCREENs 5 or 6. You do this with the CLEAR statement which, among other things, sets the size of video memory. Here's the important part: *Before using SCREEN 5 or SCREEN 6, you must enter the statement* **CLEAR ,,,32768**.

The CLEAR statement sets aside the 32K required by SCREEN 5 or SCREEN 6 as the *minimum* requirement for using these display modes. If you wish to use the active or visual page SCREEN options with SCREENs 5 or 6, you must set aside even more video memory. Furthermore, you may have to set aside video memory when using the page options for other display modes as well. (This is discussed later in the chapter.)

Setting aside 32K of video memory has its side effects, so don't clear 32K unless you need it. For one thing, you're limiting the amount of memory that can be used for other things. Secondly, CLEAR will set all numeric variables to zero and will empty ("set to null") all string variables. This can have disastrous effects if CLEAR is used in the middle of a program. Consequently, if you use CLEAR, it should be the first statement in the first line of your program. A typical graphics program that uses SCREENs 5 or 6 might start out as shown:

```
10 CLEAR ,,,32768
20 SCREEN 5
```

We've already said that CLEAR should be used at the beginning of any program to reset all variables to 0. We also mentioned that this can have disastrous effects on your program. The following program illustrates what happens to the variable A:

```
10 SCREEN 4
20 A = 2
```

```
30 PRINT A
40 FOR DELAY = 1 TO 600: NEXT DELAY
50 CLEAR ,,,32768
60 SCREEN 5
70 PRINT A
80 END
```

When you run this program, the number 2 appears in the upper left of the screen; after a few seconds, 0 appears, even though you didn't intentionally set A equal to 0. To run the program without resetting A to 0, you must move the CLEAR statement in line 50 to a new line—say line number 5—and then delete line 50. After that, the number 2 will appear in both SCREEN modes.

CLEAR is discussed further in the section of this chapter entitled "Active and Visual Pages."

Turning Color On or Off

If you are using a composite color monitor or television with your PCjr, there may be times when you prefer not to use colors. For example, if the characters on the screen are fuzzy, you can display more readable letters and numbers with the color turned off. SCREEN's color on/off option lets you do this. However, if you are using an RGB monitor, the color always remains on, even if you try to turn it off.

Color on/off can be confusing and cause difficulties because its values will have different effects in different display modes.

- In Text Mode, a value equal to zero will *disable*—turn off—the PCjr's color capabilities, and a value other than 0 (1, for instance) will *enable*—turn on—the color capabilities.
- In medium-resolution mode (display modes 1 and 4), 0 will enable the PCjr's color capabilities, while a value other than zero will disable the color.
- In the other graphic display modes (2, 3, 5, and 6), color on is always used and cannot be changed.

The following program lines produce the effects described:

10 SCREEN 0,0 sets the PCjr to Text Mode and turns off the color.

10 SCREEN 0,1 sets the PCjr to Text Mode and turns on the color.

10 SCREEN 1,0 sets the PCjr to Graphics Mode (medium resolution) and turns on the color.

10 SCREEN 1,1 sets the PCjr to Graphics Mode (medium resolution) and turns off the color.

10 SCREEN 2,1 sets the PCjr to Graphics Mode (high resolution), but the color on/off setting has no effect.

10 SCREEN 8,1 results in an error.

Since color on/off can be a variable, the following could also be used in a program:

```
10 A = 1
20 SCREEN 1,A
```

Active and Visual Pages

Think of the PCjr as a drawing tablet that consists of many different pages. The PCjr also has several "pages" on which you can draw or write. The "page" that you see on the screen is called the "visual page." The "page" that is currently available to draw on is called the "active page." The active page and the visual page aren't always the same, although they can be.

When you first turn on the PCjr, the visual and active pages are the same—page 0. That means that all activity—typing and drawing—will take place on the screen you see. SCREEN allows you to specify that page 0 is visual, but page 1 is active. Then when you draw a circle, the drawing will take place on page 1, even though you're looking at page 0. When you finally specify page 1 as the visual page, you'll instantly see the circle. To demonstrate this feature, type in and run the following two programs, one at a time. (You don't need to type in the comments at the end of the lines.) Even though you haven't been introduced to two of the commands used, CIRCLE and PAINT, don't worry about them; their function will become obvious when you run the program.

The first program draws and paints a circle in SCREEN 3 without specifying page parameters. Notice how long it takes before the complete graphics figure finally appears on the screen.

```
10 SCREEN 3  'You see and draw on page 0
20 CLS       'Clears the screen
30 CIRCLE (60,100),99    'Draws the
   circle
```

```
40 PAINT (60,100),14,15 'Paints the
   circle yellow
50 END
```

The next program does specify the page parameters, directly affecting how quickly the figure appears. First return to SCREEN 0, type **NEW** and press ENTER; then clear the screen with CLS. Next type in the following program. When you run it, the screen will clear, and for a few seconds nothing will seem to be happening; but while the screen remains blank, the circle is being drawn on a hidden page. Unlike the previous program, which drew and filled in the circle on the screen, this program shows the circle only when it is complete.

```
10 CLEAR ,,,32768    'Sets aside 32K of
   video memory
20 SCREEN 3,,1,0     'You draw on page 1
   but see page 0
30 CLS               'Clears the screen
40 CIRCLE (60,100),99    'Draws the circle
50 PAINT (60,100),14,15 'Paints the
   circle yellow
60 SCREEN 3,,1,1     'Now you see page 1
70 END
```

Because the first program did not specify a page parameter, the default values 0,0 were used. This means that the visual and the active page were the same. That is why the figure was drawn and filled in on the screen while you watched.

Now let's look more closely at the second program. In lines 20 and 60 the page parameters were specified. The drawing took place on the active page, page 1, although the screen displayed the visual page, page 0 (line 20). The drawing appeared because page 1 became the visual page in line 60. You can achieve the same results if you change line 60 to **60 END** or **60 STOP**. The circle will appear because anytime an END or STOP statement is encountered, the active page automatically becomes the visual page.

Even though SCREEN 3 doesn't normally require memory to be set aside (see the CLEAR statement in line 10), in this case it did because additional memory is needed for the additional memory pages.

Important Note! If you ever get "garbage" like vertical bars on one of the memory pages, you've probably switched from one page to another without clearing random data from the display. At best, garbage like this is confusing and irritating; at worst, it can destroy your data. The ultimate solution is to use SCREEN's *erase* option (discussed later in this chapter), but for a quick fix, isolate the page and type **CLS** ENTER, or switch screens by entering, for instance, **SCREEN 3.**

Pages can also simulate motion through a technique called "page flipping." This technique works on the same principle as drawing a figure on the edge of a page, then on the following pages, so that the same object appears at a slightly higher position each time. If you flip the pages with your thumb, the object appears to move up. (See Chapter 11 for an example of this technique.)

Determining How Many Pages Are Available

Before you can begin using pages, of course, you must be able to determine how many pages are available. This is not easy, since in different situations, such as different display modes, you will have a different number of pages. There are two factors that determine the number of available pages: the size of the current video memory and the current display mode.

As Table 3-1 shows, display modes 1 through 4 have a maximum of six pages (numbered 0-5) available, while display modes 5 and 6 have a maximum of three (0-2). The table also presents the size of each page in Kbytes. Each page in SCREEN 4, for instance, requires an additional 16K of video memory beyond the PCjr's normal needs. Therefore, additional memory must be set aside. To set aside video memory, use the CLEAR statement.

Let's say you want to use SCREEN 2 and you want to set aside three pages. Table 3-1 states that each page in display mode 2 requires 16K of video memory. To find the value required in the CLEAR statement, multiply the number of pages times the required amount of video memory. In this particular example, it would be

3 pages × 16K per page = 48K of video memory required

Since 1K of video memory is equal to 1024 bytes, you would multiply 48 times 1024 like this:

1024 × 48 = 49,152 bytes to CLEAR

Therefore, your program would begin with a statement like this one:

```
10 CLEAR ,,,49152
20 SCREEN 2
30 CLS
```

If you want to specify that page 1 was the active page and page 2 was the visual page, change line 20 so that the program will start like this:

```
10 CLEAR ,,,49152
20 SCREEN 2,,1,2
30 CLS
```

Be sure to monitor the memory you set aside. Even though you have as many as six pages (numbered 0-5) available in SCREENs 1-4 and three pages (numbered 0-2) in SCREENs 5 and 6, it isn't a good idea to set aside (CLEAR) the maximum amount of video memory, or you may not have enough memory left over to run your program.

The maximum amount of memory you can set aside with CLEAR is approximately 100K. Remember that 1 Kbyte is equal to 1024 bytes; thus, the maximum amount of memory you can reserve is 102,399 bytes (see Figure 3-3).

As Table 3-1 shows, if you multiply the maximum number of pages for each display mode by the page size, the result is always 96K (or just under 100K).

maximum number of pages × *page size* = 96K

This means that even if you reserve all of memory and use all the

Table 3-1. *Comparative Description of SCREEN Display Modes*

SCREEN Mode	Resolution	Colors Available	Maximum # of Pages	Page Size
0	Text (40-/80-column)	16	48/24	2K/4K
1	320 × 200 (medium)	4	6	16K
2	640 × 200 (high)	2	6	16K
3	160 × 200 (low)	16	6	16K
4	320 × 200 (medium)	4	6	16K
5	320 × 200 (medium)	16	3	32K
6	640 × 200 (high)	4	3	32K

CLEAR ,,,102399	CLEAR ,,,102400
Legal!	*Illegal!*

Figure 3-3. *Maximum values allowed with CLEAR to set aside memory*

pages, you still have a little memory left over for your program and related data.

Let's recap the sequence of events to follow when using pages:

1. Decide how many pages you want to use.
2. Verify the display mode you'll be in (1-4 or 5-6).
3. Multiply the number of pages times the page size (16 for SCREENs 1-4, or 32 for SCREENs 5-6).
4. Multiply the result by 1024.
5. Use the result in the CLEAR statement at the beginning of the program.

In the previous section of this chapter, we demonstrated how to draw a circle on a hidden page in SCREEN 3. When you want to use memory pages with SCREENs 5 or 6, which already require the additional 32K to be cleared out, you'll have to CLEAR an extra 32K of memory for each page past page 0. The next program does just that when it draws a circle and paints it in SCREEN 5. Notice that line 10 clears 65,536 bytes (32,768 × 2 = 65,536). (Remember—Table 3-1 stated that each page in SCREENs 5 and 6 requires 32K of memory to be set aside.)

```
10 CLEAR ,,,65536     'Sets aside 64K of
   video memory
20 SCREEN 5,,1,0,2    'You draw on page 1
   but see page 0
30 CLS                'Clears the screen
40 CIRCLE (160,100),100   'Draws the
   circle
50 PAINT (160,100),14,15  'Paints the
   circle yellow
60 SCREEN 5,,1,1    'Now you see page 1
70 END
```

Erasing Video Memory

Line 20 of the previous example used the final SCREEN operand *erase,* which allows all or part of video memory to be either protected or erased. When you change display modes, some parts of video memory are erased to make room for the new information, while other parts may be cluttered with random graphics. By including the "erase" option in the example, we prevented loss of information during the page-switching operations. To see for yourself, return to SCREEN 0, edit the program to delete the *erase* option from line 20, and run the program again. Notice how random graphics appear on the visual (but inactive) page.

In the original line 20, the value *erase* = 2 erased all of video memory, meaning that all graphics stored on any hidden pages were eliminated. This is often a good measure to take, especially at the beginning of programs.

However, there are times when you don't want memory erased. Suppose, for example, that you want to have three objects on the screen at one time. That in itself isn't a problem, but now suppose you want one of the objects created in low-resolution mode, the second in medium-resolution mode, and the third in high-resolution mode. Normally, when you create the medium-resolution object, the low-resolution object will automatically be erased. But by using the *erase* = 0 option with SCREEN when you change from one display mode to the other, you can have the three different circles displayed at once.

The default value for erase is 1. In this case, only the juncture of the old page and the new page is erased when the display mode changes. This means there may be part of both objects drawn in both modes displayed on the screen at the same time.

As previously mentioned, the last values, *erase* = 2, will erase all of video memory when the display mode changes. Only the last object drawn will be shown.

The next example demonstrates how the erase option can control random graphics. The intent of the following program is to draw two circles on the screen—each in a different SCREEN mode—and then draw a third circle in yet another mode.

```
10 SCREEN 3
20 CLS
30 CIRCLE (10,10),10
40 FOR DELAY = 1 TO 1000: NEXT DELAY
50 SCREEN 4,,,,0
```



```
60 CIRCLE (50,50),10
70 FOR DELAY = 1 TO 1000: NEXT DELAY
80 CLEAR ,,,32768
90 SCREEN 5,,,,0
100 CIRCLE (300,100),10
110 END
```

The first two circles appear as expected, but when the third circle is drawn, "garbage" also appears. The final 0 in line 90 keeps video memory from being erased, thereby allowing random graphics to appear at the same time. In this program the final 0 in line 90 is changed to a 2, all video memory is erased, and the third circle appears by itself.

```
90 SCREEN 5,,,,2
```

THE CLS STATEMENT

The CLS statement simply erases or clears the screen. Syntax 3-2 shows how to use this command.

The only function of the CLS statement is to clear the screen—it has no operands or options. The CLS command should be one of the first statements in your program; use it before you execute any commands that will display information on the screen.

If, for instance, you want to draw a circle on the screen and you run the program without first clearing the screen, the program listing and the circle will both be displayed, ruining your circle.

CLS erases or clears the screen. It has no operands.

Syntax 3-2. *The CLS Statement*

You can use CLS in either Direct Mode (type **CLS** and press ENTER) or in the Program Mode (preceded by a line number).

In any event, when the command is executed, the screen is erased and filled with the current background color, and then the cursor

returns to the "home" position on the screen.

- For Text Mode, the home position is the upper left corner of the screen.
- For Graphics Mode, the home position is the last X,Y position referenced in a program line.

In the following example, the program line will clear the screen to the current background color.

```
30 CLS
```

PRACTICING WITH SCREEN AND CLS

The following example shows how SCREEN and CLS are used together in a program. First 40-column Text Mode is selected. Next the listing is erased from the screen, the cursor (print position) located at the center of the screen, and text written on the display. After a short delay, the screen is again cleared, 4-color medium-resolution mode is selected, and a circle is drawn in the center of the screen.

```
10 SCREEN 0: WIDTH 40
20 CLS
30 LOCATE 12,10
40 PRINT "Welcome to PCjr Text Mode"
50 FOR DELAY = 1 TO 600: NEXT DELAY
60 CLS
70 SCREEN 4
80 CIRCLE (160,100),100
90 LOCATE 13,10
100 PRINT "Now it's Graphics Mode"
110 END
```

CHAPTER FOUR

Controlling Color: COLOR, PALETTE, And PALETTE USING

In addition to SCREEN, CLEAR, and CLS, two other statements are used for planning graphics programs. These statements, COLOR and PALETTE, control the color of graphics designs you've created. Individual design statements (LINE, PAINT, and so on) also allow you to control the color, but on a relatively limited scale. In other words, the colors you specify within a design statement (such as PAINT) depend on what you've previously specified with COLOR or PALETTE.

All sixteen colors are immediately available as the background color in all SCREEN modes. However, there are many factors that affect the use of colors for the foreground, including the SCREEN mode you are using, the amount of memory available, and the resolution you have selected. Since these are interrelated, the use of COLOR can become very confusing. For instance, if you are using medium resolution, you may have either 4 or 16 foreground colors available, depending on whether you are in SCREEN 4 or SCREEN 5. However, you can also select medium resolution in SCREEN 1, again with 16 available colors, but the parameters used by COLOR are entirely different from those in the other medium-resolution modes.

If you use high resolution, you have the option of either SCREEN 2 with only two colors, or SCREEN 6 with four colors available for the foreground. Once again, the COLOR statement is used differently in each of these two resolution modes.

You must also consider the amount of memory required by your program. If, for instance, you are writing a very long program and are using medium resolution, you may have to use SCREEN 4 (4 colors) instead of SCREEN 5 (16 colors) because SCREEN 5 requires more video memory than SCREEN 4 does. In this situation, you can use PALETTE to add to the limited number of foreground colors available in SCREEN 4, but much forethought and additional programming steps will be required.

These factors illustrate the direct interrelationship among the number of foreground colors, the amount of resolution, and a program's memory requirements. When writing programs, you must decide what is more important—greater resolution, more available memory, or immediate access to all 16 colors. When you have made your decision, choose the SCREEN that best suits your needs and use colors accordingly. Understanding the process requires careful study. Consequently, knowledge of this chapter is essential.

THE COLOR STATEMENT

As we discussed in Chapter 2, there are two sets of colors—background and foreground. The background color of a display can be compared to the canvas on which an artist paints; foreground colors are like the paints the artist puts on the canvas. The COLOR statement (described in Syntax 4-1) specifies which colors you want the foreground and background to be. When you power up the PCjr, the background color is always black and the foreground color white, no matter which display mode you're using. COLOR lets you change this.

The COLOR statement can be used in both Text and Graphics

COLOR *foreground, background*

foreground specifies the color you wish an object (circle, line, and so on) to be. It is a numeric expression between
1 and 15 for SCREENs 3 and 5
1 and 3 for SCREENs 4 and 6.

background specifies the color of the screen upon which objects are to be drawn. It is a numeric expression between 0 and 15.

Syntax 4-1. *The COLOR Statement*

Modes, and it performs about the same function in each. However, since we're limiting our discussion of PCjr BASIC to graphics applications, we'll only describe how COLOR is used in Graphics Mode. Even within Graphics Mode, there are some differences in the way COLOR is used.

- SCREEN 1, 4-color medium resolution, controls color differently from the other graphics display modes, so we will discuss it after SCREENs 3-6.
- SCREEN 2, 2-color high resolution, does not recognize the COLOR statement at all. If you try to use COLOR in this mode, an Illegal Function Call error will occur.
- SCREENs 3 and 5 are 16-color display modes. You can select any of the 16 colors simply by entering the COLOR statement and the code assigned to the color you want. (Table 4-1 lists colors and color codes.)
- SCREENs 4 and 6, however, are 4-color modes with only cyan, magenta, white, and black available. To access any of the other 12 colors, you must first use the PALETTE statement.

The Foreground Color

To change the foreground color, use COLOR followed by the appropriate color code. Table 4-1 lists codes for all 16 colors.

You can change the foreground color without changing the background color. (You can also change the background without affecting the foreground.) That's why we can manipulate only the foreground colors in the next few examples.

There is one limitation to foreground color: it cannot be set to color code 0. On power up, color code 0 is black. If you do want the foreground to be black, you must use the PALETTE statement to reassign colors and color codes. (For example, you could use PALETTE to assign black to code 1, and then use COLOR to indicate the foreground color.) If you try to set the foreground color to 0, an Illegal Function Call error will occur.

The Foreground Colors in SCREENs 3 and 5

As mentioned earlier, SCREENs 3 and 5 allow you to use all 16 colors without extra programming. The following short program will set the

foreground color to red, while leaving the background color black:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 COLOR 4
```

In this program, the **OK** prompt in the upper left corner of the screen turns red. To really see the effect of the foreground color change, list the program. The listing will be red.

The foreground color can also be changed in the middle of a program. The next example draws a red—the current foreground color—square on a black background. Then it draws another square in the

Table 4-1. *PCjr Colors and Color Codes*

Color Code	Color Displayed
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	Gray
8	Light gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	White

new foreground color, yellow. (Don't be concerned about the LINE command in program lines 50 and 80; it will be discussed later.)

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 COLOR 4
50 LINE (10,10)-(50,50),,BF
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 COLOR 14
80 LINE (70,70)-(110,110),,BF
90 END
```

You can use a variable for the foreground color, since it can also be a numeric expression. This next program will do this as it draws a box and flashes through all of the available colors.

```
10 SCREEN 3
20 CLS
30 C = 1
40 COLOR C
50 LINE (10,10)-(50,50),,BF
60 C = C + 1
70 IF C > 15 THEN 100
80 FOR DELAY = 1 TO 600: NEXT DELAY
90 GOTO 40
100 END
```

The Foreground Colors in SCREENs 4 and 6

Because of certain hardware and software limitations, the PCjr can't always provide all 16 colors at once. That's the case with SCREENs 4 and 6. In these modes, the PCjr gives you four colors at once, while the other 12 colors are available with additional programming.

SCREENs 4 and 6 have three available foreground colors—cyan, magenta, and white. These colors have the absolute color codes of 3, 5, and 15 respectively (see Table 4-1). But as foreground colors in SCREENs 4 and 6, they are assigned the relative position numbers 1, 2, and 3 respectively. Conceptually, in SCREENs 4 and 6 foreground colors are positions on an art palette. Figure 4-1 illustrates an art palette with four positions; three of these positions are foreground colors, and one is the background color.

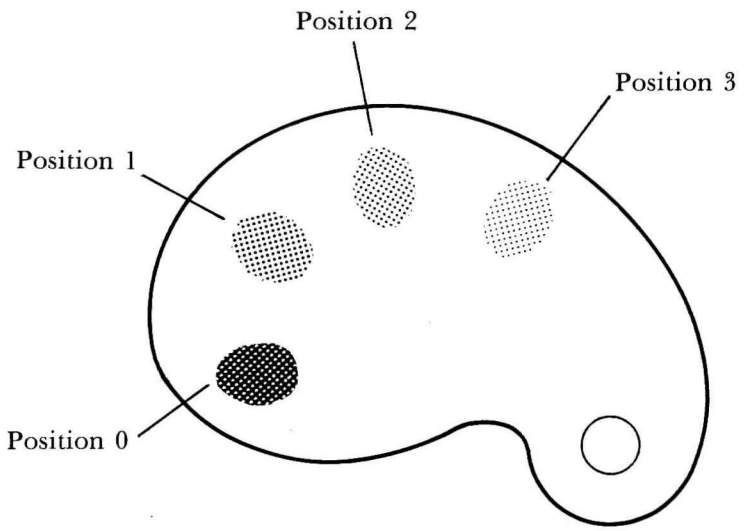


Figure 4-1. *Color positions on a palette (SCREENs 4 and 6)*

In modes 4 and 6, the PCjr refers to the palette position instead of to an individual color. In this way, any of the 16 colors can be assigned to a particular position. To select a position, and hence a color, simply include the appropriate palette position code—*not the color code as shown in Table 4-1*—with the COLOR statement. However, if you attempt to select a code greater than 3, an error will result. Figure 4-2 emphasizes that you are not selecting one of the color codes (1-15), but a position on the palette (0-3).

Remember that the foreground color can be expressed as any

```
10 SCREEN 4          10 SCREEN 4
20 CLS               20 CLS
30 COLOR 1           30 COLOR 13

Right!               Wrong!
```

Figure 4-2. *A legal and an illegal color selection in SCREEN 4*

numeric expression, including variables. Therefore, the following is also correct:

```
10 CLEAR,,,32768
20 SCREEN 6
30 CLS
40 C = 2
50 COLOR C
```

To use the other 12 colors with SCREENs 4 and 6 requires additional programming steps. Use either PALETTE or PALETTE USING statements to redefine the color codes assigned to palette positions 1, 2, and 3. (You can, for example, assign the color red to position 1 and use it as a foreground color.) As Table 4-2 and Figure 4-3 show, SCREENs 4 and 6 use palette 1.

The Background Color

The PCjr lets you change the background color to any one of 16 available colors in all of the display modes (SCREENs 0-6).

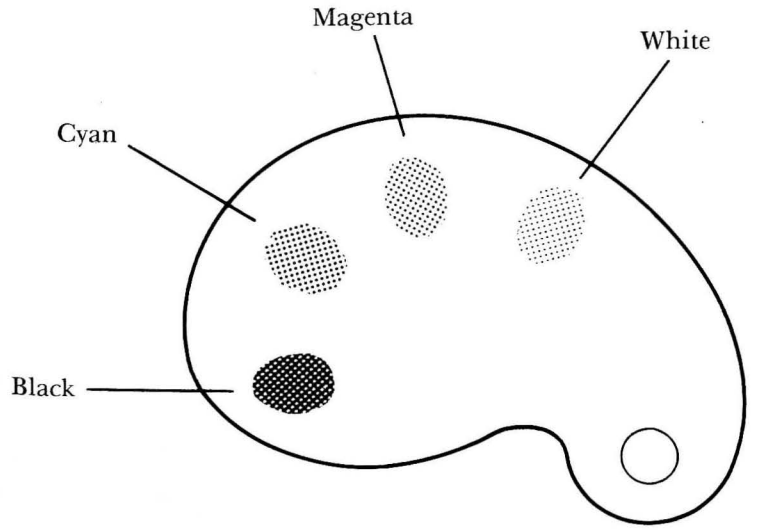


Figure 4-3. *Palette for SCREENs 4 and 6*

Table 4-2. *Default Foreground Palette for SCREENs 4 and 6*

Palette 1	
Position Code	**Color**
1	Cyan
2	Magenta
3	White

To select or change a background color, simply execute a COLOR statement. You need not specify a foreground color; the comma before the background color code will set the foreground to the default color or retain the current foreground color. The comma acts as a placeholder for the foreground color. Since *background* can be any numeric value, you could assign it to a variable and then specify a background color, as shown:

```
30 A = 4
40 COLOR ,A
```

Of course, there will be times when you want to change both the foreground and background colors. In the following program, line 40 specifies both parameters:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 COLOR 8,6
```

To illustrate how the background color can be changed independently of the foreground color, the next program draws a box near the center of the screen and fills it with the default foreground color (color code 15). Then the background changes to each available color without changing the foreground color of the box.


```
10 SCREEN 3       'Low res, 16-color
20 CLS
30 LINE (60,80)-(100,120),,BF   'Creates
   the box
40 FOR B = 1 TO 15 'B=background color
50 COLOR ,B
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 NEXT B 'Next background color
80 END
```

The COLOR Statement in SCREEN 1

In certain ways SCREEN 1 and SCREEN 4 are much the same: both are medium resolution and both are 4-color. The similarities end there, however, and you'll probably find SCREEN 1 much more awkward to work with than SCREEN 4. The PCjr probably uses SCREEN 1 (which is shown in Syntax 4-2) because it permits some degree of compatibility with existing software for the PC. Though we will describe the color capabilities of SCREEN 1, we suspect that if you're writing programs exclusively for the PCjr, you will find it much easier to use SCREEN 4 when you want a medium-resolution 4-color mode. If you are writing programs that will be used on both the PC and the PCjr, you should use SCREEN 1.

SCREEN 1 is unlike the other display modes in two ways: the background color specification precedes the foreground color in the COLOR statement; and there is a choice between *two* foreground palettes. As with SCREENs 4 and 6, you can access the other eight colors, but you'll have to use either PALETTE or PALETTE USING first.

The PCjr stores the two groups of four colors on "palettes" as shown in Figure 4-4.

The PCjr refers to one palette as palette 0 and the other as palette 1. These numbers correspond to the values you specify when using the COLOR statement. In other words, when you use COLOR in SCREEN 1, you don't specify a foreground color; instead, you select a palette and the colors it offers. When you first turn on your PCjr's power, palettes 0 and 1 use the colors listed in Table 4-3.

The PCjr always uses the color in position 3 of the default palette as the default foreground color, and black (code 0) as the default background color. Line 30 of the next program maintains the black background by specifying 0. It also specifies palette 0 for the foreground.

COLOR *background, palette*

background specifies the color of the screen upon which objects are to be drawn. It is a numeric expression between 0 and 15. *background* is optional; if omitted, 0 (black) is used.

palette specifies the foreground palette you want to use. It is a numeric expression of either 0 or 1. *palette* is optional; 1 is the default value.

Syntax 4-2. *The COLOR Statement (SCREEN 1)*

Since position 3 of palette 0 is brown, a brown foreground is displayed. You could also use red or green as foreground colors since they too are on palette 0. If you wanted to specify the color gray or any color which is not on the palette for the background or foreground, however, you would first need to use the PALETTE statement to change the colors offered on the palette.

```
10 SCREEN 1
20 CLS
30 COLOR 0,0
```

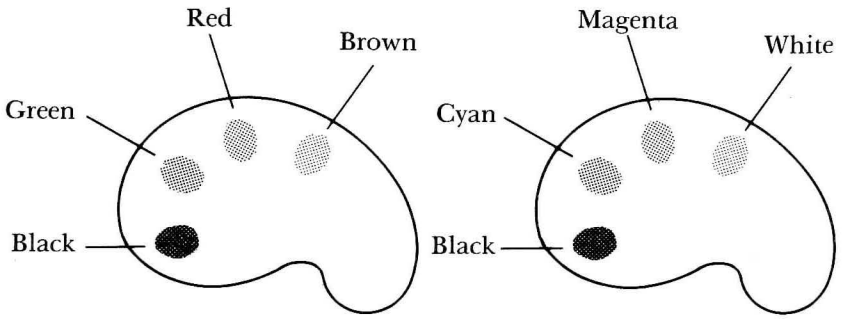


Figure 4-4. *Two palettes available for SCREEN 1*

Table 4-3. *Default Foreground Palettes for SCREEN 1*

Palette 0		Palette 1	
Position Code	**Color**	**Position Code**	**Color**
1	Green	1	Cyan
2	Red	2	Magenta
3	Brown	3	White

Program line 30 in the previous program points out a difference between the way SCREEN 1 and the other display modes handle COLOR. For one thing, the foreground palette is the second of the two operands in the statement. In SCREEN 1, the PCjr treats any numeric value after the comma as a palette position indicator. If you don't want to change the background color, leave the operand blank, include the comma, and then specify the palette position number.

When selecting a foreground palette, the PCjr will treat any odd number as 1 and any even number as 0. In Line 30 of the following program, 5 is in the foreground position. Since 5 is an odd number, palette 1 is specified. And since foreground color is always determined by palette position 3, the foreground will be white.

```
10 SCREEN 1
20 CLS
30 COLOR 0,5
```

THE PALETTE STATEMENT

COLOR controls the foreground and background colors of the display, but PALETTE determines which colors COLOR can display. In a narrow sense, PALETTE assigns color codes to position numbers on the 4-color foreground palettes. (Refer to Syntax 4-3 for a description of this command.) This is useful because SCREENs 4 and 6 allow you to access only three foreground colors at any one time (although you can access all 16 background colors). In Figure 4-1, we showed how the

PALETTE *old color, new color*

old color specifies a color code that is to be changed. It is numeric expression between 0 and 15.

new color specifies a color code that represents the replacement color and is a numeric expression between 0 and 15.

Note: If PALETTE is used by itself (no operands), the color codes will be reset to their default values.

Syntax 4-3. *The PALETTE Statement*

PCjr actually addresses a color position on a palette when you are in a 4-color mode. When you first use a 4-color mode, the background color is black (position 0) and the foreground color white (position 3). The other two foreground colors you could use are cyan (position 1) and magenta (position 2). The PALETTE statement lets you replace those colors with any of the other 12 listed in Table 4-1. When you use PALETTE in a 4-color mode, any foreground color code greater than 3 is treated as a palette position. (All numbers greater than 3 will default to 3.)

In a more general sense, however, PALETTE reassigns colors in any of the modes. This is useful for making large-scale color changes with a minimum number of program changes. In 16-color modes, color codes greater than 3 are treated strictly as color codes, not palette positions.

For instance, palette 1 has three foreground colors—cyan, magenta, and white. These colors have the *absolute* color codes of 3, 5, and 15 respectively. Table 4-1 lists absolute codes, but they are assigned the *relative* position codes of 1, 2, and 3 respectively when they are used in 4-color modes. Table 4-2 lists relative codes. If you want to change the foreground color, you don't type in the absolute color code, but the relative color position. If you want to assign a different color to a relative position code, you do so with PALETTE.

Palette Positions in SCREENs 4 and 6

Each of the four colors on the SCREEN 4 or SCREEN 6 palette is assigned to a predetermined position. When you select a foreground

color, you aren't really selecting a color; you are selecting a position on a palette and taking whatever color happens to be in that position, as Figure 4-5 illustrates.

To change a color assigned to a palette position, enter PALETTE followed by a numeric expression (number or variable) between 0 and 3. We refer to this as the *old color,* although it would be more precise to call it the "old color code." Remember, in 4-color modes the old color is usually a position as opposed to an absolute code. Next, add the "new color"—the color code that is to replace the old one on the palette. In the following program, line 50 changes the foreground palette from white (code 15) to yellow (code 14).

```
10 SCREEN 4
20 CLS
30 LINE (120,120) - (160,160),,BF
40 FOR DELAY = 1 TO 600: NEXT DELAY
50 PALETTE 3,14    'Assigns yellow (14)
   to position 3
60 END
```

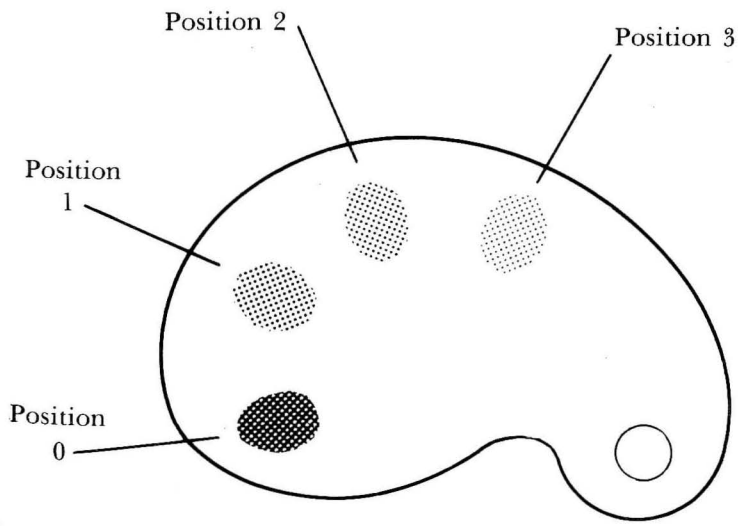


Figure 4-5. *Palette positions in SCREENs 4 and 6*

The background color can be changed just as easily. Type **PALETTE 0,3** and press ENTER to change the background from black to cyan.

PALETTE Use in SCREENs 3 and 5

The use of PALETTE in the 16-color modes is simple and straightforward. You simply specify color codes without concern for palette positions.

The next program shows the effects of changing the color codes in SCREEN 5 and demonstrates the power of the PALETTE statement. When the program asks **WHAT FOREGROUND COLOR?**, type in a number between 0 and 15. (To end the program, enter a color code greater than 15.)

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50
50 PAINT (160,100)
60 INPUT "What foreground color"; C
70 PALETTE 15,C
80 GOTO 60
90 END
```

As the above program shows, you can set the foreground color to the same color as the background color by typing **0** when asked **WHAT FOREGROUND COLOR?**. Obviously, if you do this, you won't be able to see what you're drawing, so what use could it have? You might want to make an "invisible" drawing and suddenly have it appear on the screen. (To make the foreground reappear, just type any color code other than 0 and press ENTER.)

THE PALETTE USING STATEMENT

A variation of PALETTE is PALETTE USING. This command provides a distinct advantage over PALETTE if you want to change all or part of the 16 color codes in unison. As you can imagine, it can become tedious and inefficient to continually reassign colors individually or even in a FOR-NEXT loop. PALETTE USING (described in Syntax 4-4) lets you circumvent these problems by storing the colors in an

PALETTE USING *arrayname (starting)*

arrayname is any valid array variable.

starting is the array subscript and specifies the point in the array at which color code reassignment will begin.

Syntax 4-4. *The PALETTE USING Statement*

array, then changing the contents of the array all at once.

In the following discussion, we will not explain in detail how to create arrays—only how to use them with PALETTE USING. If you're unfamiliar with arrays, review Part Six of *Hands-On BASIC*. This would be a good idea anyway, since you will need to be even more familiar with arrays later in this book.

Remember, the name you assign to the array (*arrayname*) can be any valid array variable (name) described in *Hands-On BASIC*. For example, A, W$, CAR, and PLANE are all valid array variables.

Each array variable must have a *subscript*—a number following the variable that is enclosed in parentheses. If the subscript is greater than 10 (for instance, if you redefine all 16 colors, the number will be 15), you must also use the DIM statement to dimension the array elsewhere in the program.

With the PALETTE statement, you must individually specify a current color and a replacement color. PALETTE USING does this automatically:

- The DIM statement identifies the positions (0 to 15).
- The data read into the array (usually with DATA statements) identifies the new colors to be assigned to positions.
- PALETTE USING tells the program at which position in the array to begin assigning the colors that are read in.

If PALETTE USING encounters a value of −1 when reading the data (color codes), the corresponding current color will not be changed.

The following program demonstrates this concept. When you execute the program, an array is dimensioned (line 120), and eight circles are drawn and painted. The DATA statements in lines 130 and 140 indicate which colors are to be reassigned. In this case, they are colors 1-8, the same colors in which the circles were painted. The background

color (position 0, identified by the first number in the first DATA statement) is left unchanged. PALETTE USING in line 190 then changes all of the colors automatically. If you want to change the color of the circles, change the positive values in lines 130 and 140.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 PALETTE  'Resets palette to default
40 CLS
50 DEFINT A-Z
60 X = 30: Y = 50: C = 1
70 CIRCLE (X,Y),20,8: PAINT (X,Y),C,8
80 X = X + 50: C = C + 1
90 IF X > 180 THEN GOSUB 210
100 IF C > 8 THEN 110
110 GOTO 60
120 DIM P (16) 'Sets size of the array
130 DATA -1,13,12,11,10,9,8,7     'Sets
    new colors/leaves old
140 DATA 6,-1,-1,-1,-1,-1,-1,-1
150 FOR C = 0 TO 15
160 READ P (C)
170 NEXT C
180 FOR DELAY = 1 TO 600: NEXT DELAY
190 PALETTE USING P (0) 'Changes the colors
200 END
210 X = 30: Y = 150: RETURN
```

The next program does just about the same thing, but does so "manually" by using the CIRCLE, PAINT, and PALETTE statements eight times each. You will see that it is much more cumbersome and much less efficient for complex graphics programs. The more colors you use, the more PALETTE statements will be needed, making the program overly long and complex.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 PALETTE   'Resets palette to default
40 CLS
50 CIRCLE (30,50),20,8:
   PAINT (30,50),1,8        'blue
60 CIRCLE (80,50),20,8:
   PAINT (80,50),2,8        'green
70 CIRCLE (130,50),20,8:
   PAINT (130,50),3,8       'cyan
```



```
80 CIRCLE (180,50),20,8:
   PAINT (180,50),4,8       'red
90 CIRCLE (30,150),20,8:
   PAINT (30,150),5,8       'magenta
100 CIRCLE (80,150),20,8:
   PAINT (80,150),6,8       'brown
110 CIRCLE (130,150),20,8:
   PAINT (130,150),7,8      'gray
120 CIRCLE (180,150),20,8:
    PAINT (180,150),8,8     'dark gray
130 FOR DELAY = 1 TO 600: NEXT DELAY
140 REM Palette Using does the same as
    the rest of the program
150 PALETTE 1,13
160 PALETTE 2,12
170 PALETTE 3,11
180 PALETTE 4,10
190 PALETTE 5,9
200 PALETTE 6,8
210 PALETTE 7,7
220 PALETTE 8,6
230 END
```

CHAPTER FIVE

Turning Pixels On and Off: PSET and PRESET

The previous chapters introduced several graphics statements, many of which were initializing steps; only COLOR and PALETTE actually presented visual effects. With PSET and PRESET, we begin to create and use graphics objects. PSET sets an individual picture cell called a *pixel* to a color. You can then reset the pixel with PRESET. PSET derives its name from "pixel set," which means to turn on a pixel, while PRESET comes from "pixel reset," which means to return the pixel to the previous state (assumed to be off).

Actually, the screen pixels are always turned on, at least as long as the computer and monitor are on. When you light up or turn on an individual pixel, you are really setting it to a different color. The degree of contrast between different colors indicates whether a pixel is turned on or off. When the screen is entirely black, for instance, the pixels are turned on, but set to the color black. When you then set a pixel to white, it appears that the pixel has been turned on. This is why experienced programmers usually say "set or reset a pixel" rather than "turn it on or off." As mentioned earlier, the PCjr provides you with as many as 128,000 pixels that you can set or reset.

Why is it so important to be able to set each pixel? On a broad scale, even the most elaborate PCjr graphics display is made up entirely of individual pixels. Learning how to control these pixels enables you to

create the design you want. On a more practical level, many lines automatically created with the LINE or CIRCLE statements are too straight for detailed drawing. PSET is more flexible, allowing you to create a line with sharp curves and bends. Furthermore, you may remember that the PCjr has an X,Y coordinate specification feature called the *last referenced point (LRP).* This feature lets you specify a screen point for graphics action (drawing, painting, and so on) to begin automatically. Consequently, you don't have to include an absolute screen coordinate within the action command itself. The advantage is that when you need to make large-scale program changes, you only have to alter a single program line instead of several. PSET and PRESET are ideal for identifying LRPs.

PSET and PRESET involve two concepts discussed earlier:

- The X,Y coordinate system
- Color, particularly the foreground color.

If you're unfamiliar with either or both of these, review Chapters 1 and 2 for a general discussion of PCjr graphics and Chapter 4 for a complete description of PCjr color.

You will recall that in the X,Y coordinate system, the X-direction is across the screen (horizontal) and the Y-direction is down the screen (vertical). Unless you've redefined the screen coordinate system, always start counting at X=0, Y=0 (referred to as 0,0), which is in the upper left corner of the display in all graphics modes.

THE PSET STATEMENT

Remember that when you enter BASICA, the PCjr uses color code 0 as the default background color. (Unless you use PALETTE to specify otherwise, black is assigned to code 0.) The default foreground color is always equal to the highest available color code. For instance, if you are in SCREEN 5 (medium resolution, 16-color), the default foreground color will be code 15—white. If you are in SCREEN 6 (high resolution, 4-color), the default foreground color will be code 3 for palette 1—white again. Syntax 5-1 describes PSET.

PSET *(x,y), color*

PSET STEP *(x,y), color*

x specifies the X-axis (horizontal) coordinate of a display pixel. It is a numeric expression between
0 and 199 for SCREEN 3
0 and 319 for SCREENs 1, 4, and 5
0 and 639 for SCREENs 2 and 6.

y specifies the Y-axis (vertical) coordinate of a display pixel. It is a numeric expression between 0 and 199 in all graphics display modes.

color specifies the color you want the pixel to be. It is a numeric expression between
0 and 3 in SCREENs 1, 4, and 6
0 and 15 in SCREENs 3 and 5
0 or 1 in SCREEN 2.

color is optional; if omitted, the highest available color code (current foreground) will be used.

STEP indicates the location of the specified *x,y* position is relative to the last referenced point instead of an absolute screen location.

Syntax 5-1. *The PSET Statement*

The X-Axis Coordinate

To PSET a pixel, you must first specify its X-axis position. Remember that on this axis, all counting begins on the left side of the screen at 0 and progresses left to right.

In high-resolution Graphics Mode, the PCjr screen has 640 individual pixels across the screen, numbered from 0 to 639. In medium resolution, there are 320 (0 to 319) pixels horizontally, while in low resolution there are 160 (0 to 159). Figure 5-1 describes the relative pixel width of each pixel on the X-axis.

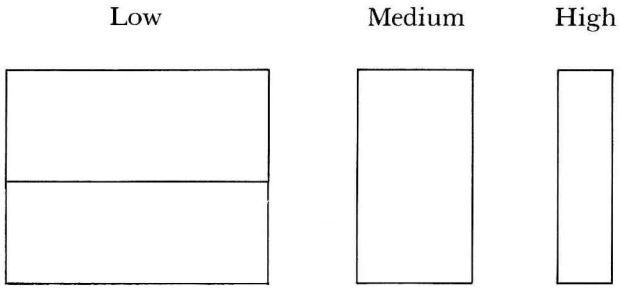


Figure 5-1. *Comparison of relative pixel size in different resolutions*

The Y-Axis Coordinate

Once you have identified the X-position of the pixel you want to turn on, you must identify the Y-coordinate. To do this, begin counting down the X-axis until you reach the appropriate Y-axis location. In all three resolutions, there are 200 rows (numbered 0 to 199). Zero (0) is at the top and 199 is at the bottom of the screen.

Identifying X,Y Coordinates

To identify the X,Y position of 10,10, for instance, just count ten pixels over on the X-axis beginning at 0,0, then count down ten more, as Figure 5-2 illustrates.

Once you've identified a pixel like 10,10, use it in a statement as shown:

```
30 PSET (10,10)
```

Remember that in medium resolution, X can be any value up to 319 (0 is included in the range) and Y up to 199. Therefore, the following short program will set the last available pixel on the screen (the lower

right corner of the screen). If you can't see the pixel, check the horizontal adjustment on your video monitor.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 PSET (319,199)
50 END
```

If you increase either the X- or Y-value—even by 1—the pixel identified will be off the screen and you won't see it. Figure 5-3 illustrates the location of the pixel.

To set the pixel in the exact middle of the screen, divide by 2 both the maximum X- and Y-values for the display mode you are using. In

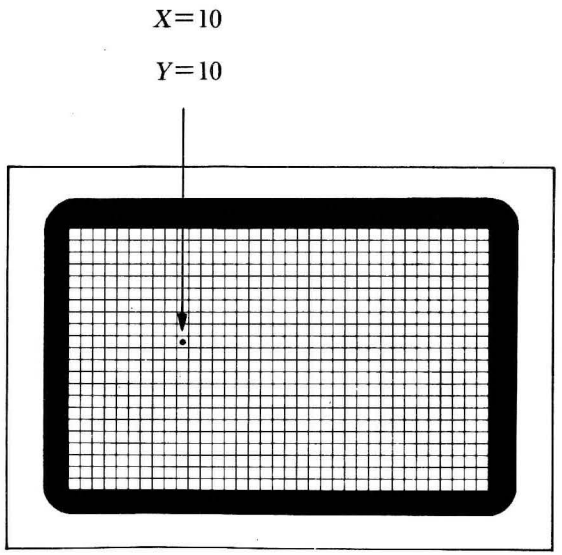


Figure 5-2. *Identifying a pixel by its X,Y axis*

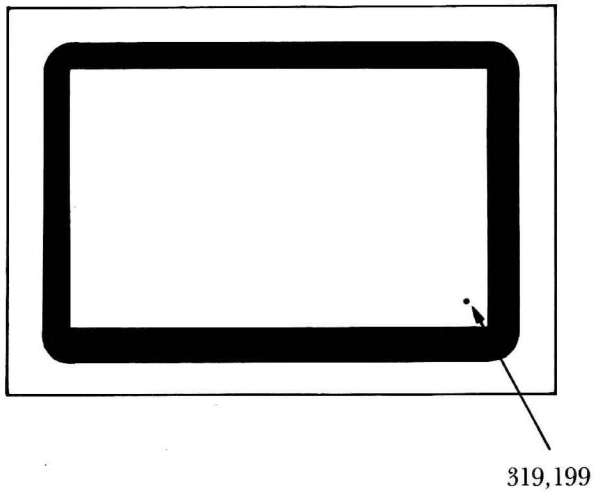


Figure 5-3. *Pixel 319,199 (medium resolution)*

medium resolution, this would be 320 and 200 by 2; that produces the X,Y value of 160,100 used in a program line like this one:

```
30 PSET (160,100)
```

You can specify an X,Y value that is out of the allowable range for the resolution you're using, and an error will not occur. However, you will not be able to see if the pixel is set or reset, since it will be off the visible screen. In low resolution, for example, you can specify a pixel at X,Y coordinate 1000,1000 and an error will not occur; obviously, you won't see this pixel light up.

Setting the Pixel Color

In the examples used so far, the color of the pixel has been the default color. Except for SCREEN 2 (2-color, high resolution), PSET's color option also lets you control the color of the pixel.

Notice that the color option must be outside the X,Y parentheses and preceded by a comma. The following program will set a green dot in the middle of the screen. (Green is color code 2.)

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 PSET (160,100),2
50 END
```

If the pixel was too small for you to see, try this program, which uses a lower resolution to create a larger dot. Notice that the X,Y coordinates have changed while the color code is the same.

```
10 SCREEN 3
20 CLS
30 PSET (80,100),2
40 END
```

In the following example, the pixel in the middle of the display is set to magenta. Remember that SCREEN 4 is a 4-color mode, and magenta is in position 2 of SCREEN 4's palette.

```
10 SCREEN 4
20 CLS
30 PSET (160,100),2
40 END
```

Since the color option, like most PCjr operands, can be a variable as well as a number, you can include statements like the following in programs:

```
10 SCREEN 4
20 CLS
30 C = 2
40 PSET (160,100),C
50 END
```

Sample PSET Program

The examples in this chapter emphasize that pixel-by-pixel manipulation lets you create almost any design you wish. All of the graphics

commands described later in this book (LINE, CIRCLE, PAINT, and so forth) are nothing more than sophisticated applications of PSET; anything that can be done with them can be done with PSET.

For example, the following program uses PSET to create a flower that has eight petals. While the program does use a couple of PCjr mathematical functions in addition to a FOR-NEXT loop and GOTO statement, the PSET statement actually creates the design.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 PI = 3.14159
50 J1 = 0: J2 = 2 * PI
60 A = 360: B = 50
70 X = (J2-J1)/A
80 FOR C = J1 TO J2 STEP X
90 R = B * COS (4 * C)
100 X = R * SIN(C)
110 Y = R * COS(C)
120 PSET (160 + X, 100 + Y),2
130 NEXT C
140 END
```

Relative Moves and the STEP Option

There are two different ways to specify X,Y coordinates for PSET. One way is the *absolute* method (the method described until now), in which you specify an exact point on the display. The other method is the *relative* method, in which you indicate a coordinate that is "so many pixels away from the current or last referenced pixel."

When using STEP for relative positioning capability, you must begin thinking of locations that are greater than or less than the last referenced or current point. Figure 5-4 illustrates this concept.

For instance, suppose you set a pixel in the middle of the screen like this—**PSET (160,100)**. Next you want to move ten pixels to the right on the X-axis and ten more spaces down on the Y-axis, but you don't know that particular absolute coordinate position. To use a relative move, you would only need to enter the statement **PSET STEP (10,10)**. If you wanted to specify a position that was ten spaces up from the center and ten spaces to the right, you would use a statement like this—**PSET STEP (−10,10)**.

The following program is an example of how you can perform relative moves with PSET. (We are using SCREEN 3 because the pixels are larger and easier to see.) The points in Figure 5-5 are referenced in the program's comment lines.

```
10 SCREEN 3
20 CLS
30 PSET (80,100) 'Sets the absolute
   point
40 FOR DELAY = 1 TO 600: NEXT DELAY
50 PSET STEP (-10,-10)   'See Point A
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 PSET (80,100)   'Sets absolute point
   again
80 PSET STEP (10,10) 'See Point B
90 FOR DELAY = 1 TO 600: NEXT DELAY
100 PSET (80,100)    'Sets absolute
    point again
110 PSET STEP (-10,10) 'See Point C
120 FOR DELAY = 1 TO 600: NEXT DELAY
```

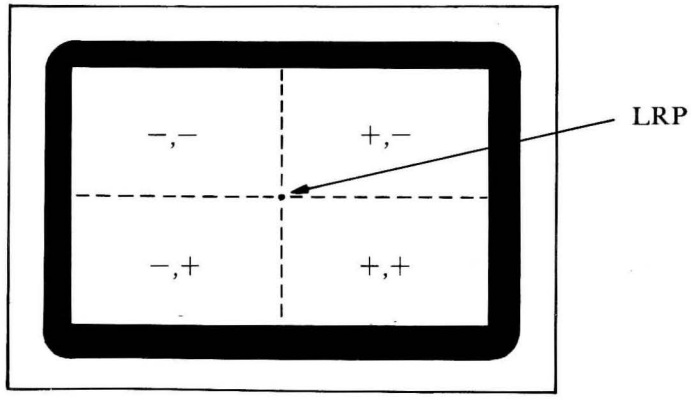


Figure 5-4. *Values indicating direction from last referenced point for relative moves*

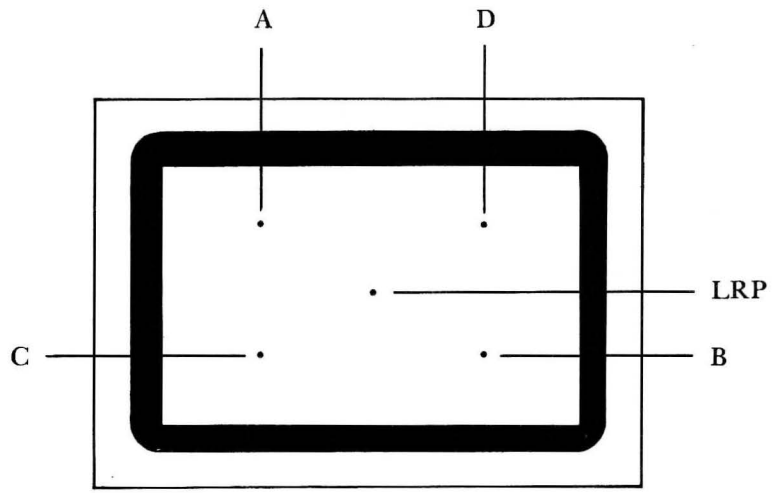


Figure 5-5. *Pixels relative to an absolute pixel*

```
130 PSET (80,100)     'Sets absolute
    point again
140 PSET STEP (10,-10) 'See Point D
150 END
```

In this example, we always took the last referenced point back to the same absolute coordinate (the center of the screen). However, that certainly doesn't have to be the case. In the next short example we move from one relative point to the next.

```
10 SCREEN 3
20 CLS
30 PSET (80,100) 'Sets first point,
   absolute
40 FOR DELAY = 1 TO 600: NEXT DELAY
50 PSET STEP (10,-10)  'To the right and
   up
60 FOR DELAY = 1 TO 600: NEXT DELAY
```



```
70 PSET STEP (50,50)    'To the right and
   down
80 FOR DELAY = 1 TO 600: NEXT DELAY
90 PSET STEP (-40,22) 'To the left and
   down
100 END
```

In the first part of this chapter, we mentioned that PSET is ideal to use when specifying a LRP for commands like CIRCLE and LINE. To illustrate this, the next example uses PSET to specify an X,Y coordinate and CIRCLE to draw a point relative to that absolute pixel. All of the following circles are then drawn relative to the previous relative point. Notice that the green PSET pixel is set just before the next circle is drawn. If you change either of the X,Y values in line 60 to a negative, the circles will be drawn at different locations.

```
10 SCREEN 3
20 CLS
30 X = 20: Y = 20
40 PSET (X,Y),2
50 FOR DELAY = 1 TO 600: NEXT DELAY
60 CIRCLE STEP (33,44),20
70 FOR DELAY = 1 TO 600: NEXT DELAY
80 X = X + 10: Y = Y + 10
90 IF X > 120 THEN 110
100 GOTO 40
110 END
```

THE PRESET STATEMENT

PRESET (see Syntax 5-2) lets you reset any pixel that has been set. It doesn't matter how the pixel was set, whether by PSET, LINE, CIRCLE, or any of the graphics action commands. PRESET will reset any pixel to the background color.

The syntax of PRESET is almost exactly like that of PSET. The X,Y coordinate system works in exactly the same way for PRESET as it does for PSET. In this section, we will show you how PRESET resets a pixel.

If you use PRESET's color option, PRESET will behave and perform exactly like PSET. In most cases, however, you'll use PRESET to reset a pixel that you set with PSET.

PRESET *(x,y), color*

PRESET STEP *(x,y), color*

x specifies the X-axis (horizontal) coordinate of a display pixel. It is a numeric expression between
0 and 199 for SCREEN 3
0 and 319 for SCREENs 1, 4, and 5
0 and 639 for SCREENs 2 and 6

y specifies the Y-axis (vertical) coordinate of a display pixel. It is a numeric expression between
0 and 199 in all graphics display modes.

color specifies the color you want the pixel to be. It is a numeric expression between
0 and 3 in SCREENs 1, 4, and 6
0 and 15 in SCREENs 3 and 5
0 or 1 in SCREEN 2

color is optional; if omitted, 0 is used.

STEP indicates that the *(x,y)* value will be a relative move as opposed to an absolute move.

Syntax 5-2. *The PRESET Statement*

The only difference in the color option is that if you don't specify a color, the PCjr will automatically use color code 0. The default color for this code is black.

Sample Uses of PRESET

The following program uses PSET and PRESET to create a flashing pixel in the middle of the screen. Press BREAK to stop the program.

```
10 SCREEN 3
20 CLS
30  PSET  (80,100),2
40 FOR DELAY = 1 TO 600: NEXT DELAY
50  PRESET  (80,100)
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 GOTO 30
```


Relative Moves and the STEP Option

Like PSET, PRESET can use the STEP option to generate relative moves. When using either of the two statements starting relative moves, PRESET offers the advantage of not displaying the LRP pixel. In an earlier example, PSET set a green pixel, and a circle was drawn relative to it. If you did not want to display the green pixel, just the circles, you could have used PRESET instead of PSET to identify the LRP.

The following program draws a circle, paints it red, and then uses PRESET to reset a line across the circle. The line is black, the current background color. Press BREAK to end the program.

```
10 SCREEN 3
20 CLS
30 CIRCLE (80,100),99
40 PAINT (80,100),4,15
50 PRESET (20,20)     'Identifies the LRP
60 PRESET STEP (1,1) 'Draws the line
70 GOTO 60
```

If program line 50 was not included in the program, the LRP will be the center of the screen, as identified in line 40. Therefore, the line will begin at 80,100. This is a handy way to draw radius lines in a circle. Delete line 50 and run the program to see.

The last program in this chapter lets you use the PCjr screen as a drawing board. It starts with a dot set in the center of the screen. You can then draw to the left, right, up, or down by pressing the respective arrow keys on the right side of the keyboard. You can also change the color of the line you are drawing. If you want to pause a line (before changing directions or color, for instance), press the SPACEBAR. To end the program, press Q (for quit).

```
10 CLEAR ,,,32768!
20 SCREEN 5
30 CLS
```

```
40 KEY (11) ON: KEY (12) ON:
   KEY (13) ON: KEY (14) ON
50 INPUT "What color of line (1-15)";C
60 X1 = 160: Y1 = 100: X2 = 0: Y2 = 0
70 A$ = INKEY$
80 IF A$ = "Q" OR A$ = "q" THEN END
90 IF A$ = " " THEN X2 = 0: Y2 = 0
100 ON KEY (11) GOSUB 1100:
    ON KEY (12) GOSUB 1110:
    ON KEY (13) GOSUB 1120:
    ON KEY (14) GOSUB 1130
110 X1 = X1 + X2: Y1 = Y1 + Y2:
    IF X1 < 0 THEN X1 = 0
120 IF X1 > 300 THEN X1 = 300
130 IF Y1 < 0 THEN Y1 = 0
140 IF A = 1 THEN PSET (X1,Y1)
150 PSET (X1,Y1),C
160 GOTO 70
170 END
1100 Y2 = -1: X2 = 0: RETURN
1110 X2 = -1: Y2 = 0: RETURN
1120 X2 = 1: Y2 = 0: RETURN
1130 Y2 = 1: X2 = 0: RETURN
```

CHAPTER SIX

Creating Lines And Rectangles: LINE

One of the most powerful PCjr graphics commands you will use is the LINE statement. This single statement can produce a variety of graphics designs, including a straight line between two points, a box (square or rectangle), and a dotted or dashed line in a variety of patterns. Boxes created with LINE can also be automatically filled with any available color.

Before you begin working with this statement, however, be sure that you're familiar with the PCjr's screen coordinate system, because this command (like most of the graphics statements) requires you to identify specific X,Y points on the display.

The LINE statement is like a connect-the-dots picture in a child's coloring book: once dots are identified, you draw straight lines connecting them. Figure 6-1 illustrates this simple concept.

LINE, however, requires you to identify the dots (pixels) as well as connect them. There are two ways to identify pixels with PCjr graphics: the absolute method and the relative method. With the absolute method, you simply state the X,Y position of the pixel you want to use. With the relative method, however, the "next" pixel is identified in relation to the "previous" one, the last referenced point or LRP.

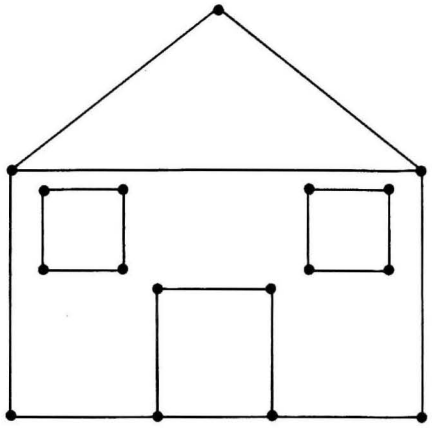


Figure 6-1. *Basic premise of LINE command*

THE LINE STATEMENT

Any line created by LINE is simply a series of pixels, just like the dots you set with PSET. To draw a line using PSET, however, you need to set each pixel individually (usually with a FOR-NEXT loop). LINE does this automatically. As Syntax 6-1 shows, LINE is a powerful and complex statement, which means that it can be difficult to understand. We will start our discussion of this statement by describing its simplest use—drawing a straight line.

The Starting Point

Figure 6-2 shows how to identify the starting point (pixel) for a line by counting over on the X-axis, then down on the Y-axis. Once you have located the line's starting point, type in the word **LINE**, followed by the X,Y value of the pixel location inside parentheses, as shown: 50 LINE (10,10). If you've previously identified the X,Y values as variables, they can also identify the starting point.

LINE (*x1,y1*)–(*x2,y2*), *color*, **B** or **BF**, *style*
LINE–STEP (*x2,y2*), *color*, **B** or **BF**, *style*

x1 and *y1* specify the line's starting point. *x1* is a numeric expression between
0 and 159 (SCREEN 3),
0 and 319 (SCREENs 1, 4, and 5), and
0 and 639 (SCREENs 2 and 6).

y1 is a numeric expression between 0 and 199 for all resolutions. *x1* and *y1* are optional; if omitted, the center of the screen or the last referenced point (LRP) is used.

x2 and *y2* specify the line's ending point. *x2* is a numeric expression between:
0 and 159 (SCREEN 3),
0 and 319 (SCREENs 1, 4, and 5), and
0 and 639 (SCREENs 2 and 6).

y2 is a numeric expression from 0 to 199 for all resolutions.
color specifies the color you want the line to be. It is a numeric expression between:
0 and 3 in SCREENs 1, 4, and 6,
0 and 15 in SCREENs 3 and 5, and either
0 or 1 in SCREEN 2.

color is optional; if omitted, the highest available color code will be used.
B or **BF** indicates that you want a box drawn with the start and end points as diagonals. If **BF** is used, the box will be filled with the color specified by *color*. **B** and **BF** are optional; if omitted, a straight line will be drawn.

style specifies the pattern of a line (dotted, dashed, and so on). It is a either a positive or negative numeric expression.

style is optional; if omitted, a solid line is used. If **BF** is used, *style* may not be.

STEP specifies a starting point relative to the last referenced point. **STEP** is optional; if omitted, the center of the screen or the LRP is used.

Syntax 6-1. *The LINE Statement*

```
20 X = 50
30 Y = 100
40 LINE (X,Y)
```

However, the starting point doesn't have to be used in every instance. When you don't specify a line's starting point, the PCjr follows one of two rules: either the line starts at the last referenced point (LRP), or the line starts at the center of the screen if a previous pixel has not been specified. In cases like these, all you have to do is identify the ending point. That ending point can be absolute or relative, but in both cases, be sure to preface it with a minus (–) sign.

The Ending Point

The ending point of a line is also an X,Y coordinate. It is important to note that a minus sign (–) must be in front of the ending point's X,Y value; for instance, 10 LINE (10,10) – (20,20). Since the line doesn't have to be drawn from left to right, the value for the ending point can be less than that of the starting point, as in 30 LINE (100,100) – (70,80). You now have enough information to draw your first line.

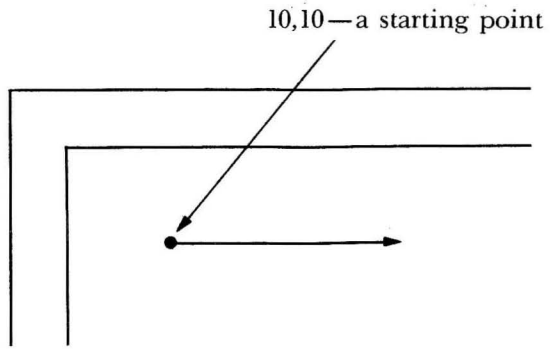


Figure 6-2. *Pixel 10,10 used as the starting point of a line*

Type in the following short program to try the LINE statement:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 COLOR 3,14
50 LINE (10,10) - (160,100)
60 END
```

When you type **RUN** and press ENTER:

- The PCjr is in medium resolution.
- The screen turns yellow.
- A cyan line is drawn from the upper left corner to the center of the screen.

Go ahead and experiment a bit. Try changing the values of the starting and ending points to draw lines in other places on the screen. If you want to draw multiple lines, insert additional LINE statements using line numbers 42, 45, 47, and so on. Now you see why it's useful to increment your program line numbers by 10—it makes adding new lines easier. Since you are already familiar with COLOR, try changing the color of the background and the line itself (the foreground).

As we mentioned, the starting point is optional, but you must identify where the line ends.

If you don't specify the line's starting point, the PCjr will begin drawing the line at the ending point of the previous line or at the last referenced point (LRP) of any other graphics statement. If you haven't already used LINE, the PCjr will use the center of the screen. The following short program draws a horizontal line that has absolute starting and ending points, and then draws a vertical line, even though this line's starting point isn't included. Finally, a third line is drawn, again without your typing in a starting location. See Figure 6-3.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE (50,10)-(100,10)
50 LINE -(100,60)
60 LINE -(150,60)
70 END
```

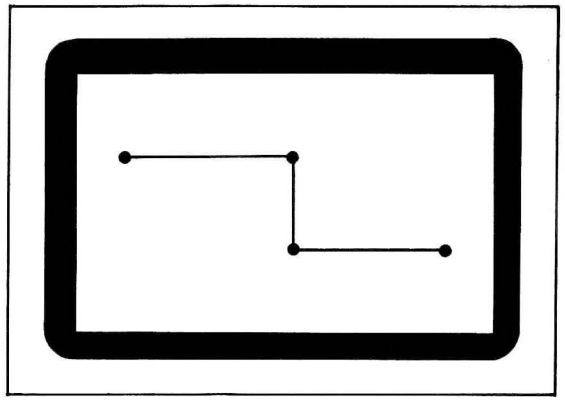


Figure 6-3. *Sample lines using one absolute and two relative starting points*

If you decided not to specify an absolute starting location in the entire program, as in the following program, the drawing action will begin at the center of the screen. Figure 6-4 shows what the lines will look like.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE -(100,10)
50 LINE -(100,60)
60 LINE -(150,60)
70 END
```

Controlling a Line's Color

As you probably noticed, if you don't use LINE's color option, the PCjr defaults to the highest available color code. In SCREENs 4 and 6, color code 3 will be used; in SCREENs 3 and 5, 15 will be used. For instance, the following program draws a white line on a black background. (Remember, black—code 0—is the default background for all

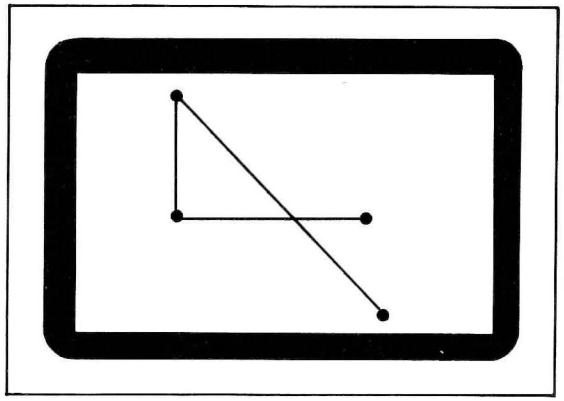


Figure 6-4. *Sample line using relative starting points only*

screen modes, and since the highest color code available in SCREEN 3 is 15, white is the default foreground.)

```
10 SCREEN 3
20 CLS
30 LINE (10,10)-(100,100)
40 END
```

Since the above program used SCREEN 3 (a 16-color mode), you can select any of the available colors by adding LINE's color option in program line 30. To draw a red line on a black background, use a line like this one:

```
30 LINE (10,10)-(100,100),4
```

However, in SCREENs 1, 4, or 6, the only available foreground colors upon entering Graphics Mode are cyan, magenta, and white (position codes 1, 2, and 3). The following program draws a high-resolution, magenta line on a black background.

```
10 CLEAR,,,32768
20 SCREEN 6
30 CLS
40 LINE (50,10)-(100,10),2
50 END
```

If you are using a 4-color mode and you want to use a color other than cyan, magenta, or white, you'll need to use PALETTE to reassign the color codes 1 through 3, as in the following program.

```
10 CLEAR
20 SCREEN 6
30 CLS
40 PALETTE 2,14
50 LINE(10,10)-(50,50),2
60 END
```

Just as with almost every other PCjr statement or function, you can use a variable to select the color of a line. The following program does this and more. By using variables and the RND (random) function, different-colored lines of different lengths will appear randomly around the screen.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 INPUT "How many random lines do you
   want drawn"; NUMBER
50 CLS
60 FOR NUMBER = 1 TO NUMBER
70 X1 = 300 * RND
80 Y1 = 190 * RND
90 X2 = 300 * RND
100 Y2 = 190 * RND
110 C = 15 * RND + 1
120 LINE (X1,Y1) - (X2,Y2),C
130 FOR DELAY = 1 TO 600: NEXT DELAY
140 NEXT NUMBER
150 END
```

When you run this program, you'll be asked **HOW MANY RANDOM LINES DO YOU WANT DRAWN?**. After that, a line of a random length, location, and color will be drawn on the screen, followed by

another line, and so on. Other than the number of lines, the only predictable aspect of the program is the range of the line's starting and ending points.

Creating a Box

One way to make a box or rectangle is to use four LINE statements. While this gets the job done, it isn't very efficient programming, especially since the PCjr provides an easier way. LINE's B (box) option enables you to create a box with just a single statement.

Unlike other LINE operands, B does not represent a numeric expression; it must be typed in as is. When the B option is used, the starting point ($x1$,$y1$) becomes the upper left corner of the box, while the ending point ($x2$,$y2$) becomes the lower right corner (diagonal points, in short). See Figure 6-5.

The following program will create a box that takes up most of the screen. Notice that the background is black and the foreground—the color of the box—is green.

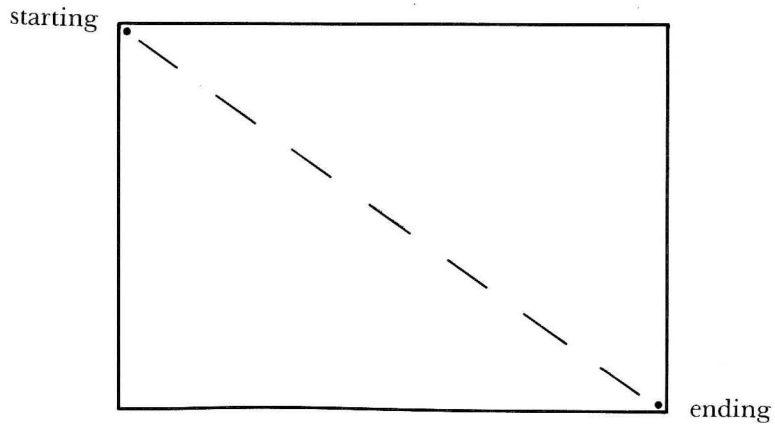


Figure 6-5. *Diagonal points when using LINE to draw a box*

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE (10,10)-(310,190),2,B
50 END
```

It's as simple as that. In the previous section, we included a program that drew random lines on the screen. You can create random boxes simply by adding ,B to the end of line 120. (You might also want to change the prompt.)

```
120 LINE (X1,Y1) - (X2,Y2),C,B
```

Now when you run the program, different-colored boxes will appear at different points around the screen.

The following program shows you another way to create random boxes. Press BREAK to end the program.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE -(300 * RND, 200 * RND), 15 * RND + 1, B
50 GOTO 40
```

The next program creates 11 boxes of diminishing sizes. The last box drawn is a straight line. Try experimenting with the program. Draw more than 11 boxes and see what happens.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 NUMBER = 1   'Starts the counter for
   number of boxes
50 X1 = 0: Y1 = 0  'Upper-left diagonal
60 X2 = 300: Y2 = 200 'Lower-right
   diagonal
70 C = 15 * RND + 1   'Random colors
80 LINE (X1,Y1) - (X2,Y2),C,B  'Draws
   box
90 X1 =X1 + 10: Y1 = Y1 + 10   'Moves
   upper-left to center
100 X2 = X2 - 10: Y2 = Y2 - 10 'Moves
    lower-right to center
```



```
110 NUMBER = NUMBER + 1: IF NUMBER > 11
    THEN 130
120 GOTO 70
130 END
```

Filling a Box With Color

In addition to creating a box with B, you can fill that box with the foreground color using LINE's BF ("box fill") option. Remember that in the 4-color display modes, this color is selected by the color option working in conjunction with the PALETTE statement. If you don't indicate which color you want, the PCjr will use the default colors. The only rule governing the use of BF is that you can't use it and the *line style* option in the same statement. The following program uses the BF option:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE (100,100) - (150,150),4,BF
50 END
```

In this example, the screen's background color is set to black, and a box is drawn in the lower half of your screen. When the box is complete, it is filled with red (color code 4).

A previous example showed you how to create random boxes on the screen. The following program adds an F to the LINE statement to fill the boxes randomly. Press BREAK to end the program.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE -(300 * RND, 200 * RND), 15 *
   RND + 1, BF
60 GOTO 40
```

Creating Different Types of Lines (Line Styles)

So far, all of the lines you have made have been solid lines. The PCjr, however, lets you create dotted or dashed lines (or a combination of the

two). Different line styles are especially useful with line charts, since you can tell the difference between two different lines at a glance.

It may be hard to believe, but there are over 65,000 different types of dotted lines you can draw. However, the difference is so minute among most of them that you will never be able to see it by simply looking at the screen.

How is it possible to have so many line styles? Remember, we explained that each dot on the screen is called a pixel and that every pixel can be either set or reset. Let's say you want to create a dashed line consisting of dashes of four pixels each, with the same distance between each dash.

— — — —

In PCjr graphics terms, you would do this by turning four pixels on, then four off, and so on:

on on on on off off off off on on on on off off off off

The binary (number base 2) 1 sets a pixel to on and the number 0 resets it to off. So in binary terms, you could set four and reset four, like **1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0**. While this is easy to remember, it's awkward to work with, especially when you want line style 65,533 (which would be a lot of 0s and 1s). Consequently, the PCjr lets you specify the line style you want by using the hexadecimal (base 16) numbering system. Until you get used to it, hexadecimal may be more difficult to use, but after a while you shouldn't have any trouble with it.

Instead of the 1s and 0s used to turn on and off four pixels at a time in the above example, you could simply use the hexadecimal number: **F0**. As you can see, that is much easier to type in. Since you may not be familiar with hexadecimal numbers, Appendix A lists decimal, hexadecimal, and binary equivalent numbers. For convenience, think of the binary number as pixels on and off (0s and 1s), then see what hexadecimal number to use.

There is one more thing to be aware of: you must tell the PCjr that you are going to use a hexadecimal number. How do you indicate this? Simply enter the characters **&H**. The PCjr will then treat anything that follows as a hexadecimal number, as long as it is within the allowable range (that is, not greater than the decimal number 65355). For instance, to draw a line of dashes that uses eight pixels for each dash

and eight pixels between each dash, as illustrated in Figure 6-6, use the following version of the LINE statement:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 LINE (10,100) - (300,100),,,&H00FF
50 END
```

When this program is executed, the line will begin at the left and go across the screen.

Table 6-1 provides a few commonly used line styles along with their binary (pixel-by-pixel) interpretation and the hex number that generates them.

There are two things to remember when using the codes in Table 6-1: First, that every hexadecimal number must be preceded by &H to tell the PCjr that a hex number will follow; and second, that you should think of each 0 and 1 as a pixel that is either turned on or off. By writing down on a piece of paper the dot pattern (line style) you want, you can easily determine which binary number represents the style, then use the conversion charts in the appendixes to convert that binary number to a hexadecimal number.

For instance, if you want to have a dotted line that is 1 pixel on and 3 pixels off, then another 1 pixel on and 3 more off, and so on, write it down on a piece of paper as **1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0**. If you look at the conversion charts in the appendix, you'll find this binary number listed in the binary column. Next to it will be a hexadecimal column where you will find 8888 (and next to that will be a decimal column

— — — — — — — —

Figure 6-6. *Sample line of dashes*

Table 6-1. *Sample Line Styles and Codes to Create Them*

Line Style	Pixel On/Off (Binary)	Hexadecimal Numbers
Dot-dash	1000 1111 1111 1000	8FF8
Short dash	1111 0000 1111 0000	F0F0
Long dash	0000 0000 1111 0000	00FF
Solid line	1111 1111 1111 1111	FFFF
Wide dots	0000 1000 0000 1000	0808
Medium dots	1000 1000 1000 1000	8888

with the decimal equivalent of 136). Use the hex equivalent to specify the line style.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 LINE (160,100)-(100,100),4,,&H8888
50 END
```

When you run this program, a red line (color code 4) of medium dots (&H8888) will be drawn starting from the center of the screen.

As we mentioned, different line styles are extremely useful when constructing line charts or graphs that need to differentiate one line from another. For an excellent example of this application, see Chapter 16 of this book.

The following example draws a box to illustrate the different line styles. When you are prompted, **WHAT LINE STYLE DO YOU WANT TO SEE?**, type in a hexadecimal number between &H0 and &HFFFF. (Be sure to include the prefix &H to any number you type in.) That style of line will then be used to draw the box. Press any key to be prompted for another box and line style; press BREAK to end the program. When you run the program, try for contrast in line styles—&HFFFF is a solid line, &H0 is a blank line. For a list of hexadecimal numbers, see Appendix A.


```
10 SCREEN 0
20 CLS
30 INPUT "What line style do you want to
   see?"; H
40 SCREEN 3
50 CLS
60 LINE (30,30) - (130,130),,B,H
70 A$ = INKEY$: IF A$ = "" THEN 70
80 GOTO 10
```

Relative Moves and the STEP Option

The one option we haven't discussed yet, STEP, lets you use relative positioning to identify a line's starting point. Remember that there are two different ways to specify X,Y coordinates for LINE. One is the *absolute* method we've been describing in the previous examples. The other is the *relative* method in which you indicate a coordinate that is "so many pixels away from the current or last referenced pixel."

When using the STEP option's relative positioning capability, you must begin thinking of locations that are greater than or less than the last referenced or current point. See Figure 6-7.

For instance, suppose you end a line in the middle of the screen like this—**LINE (10,10)–(160,100)**. Next you want to end the next line ten pixels to the right on the X-axis and ten more spaces down on the Y-axis, but you don't know that particular absolute coordinate position. To use a relative move, you would only need to enter the statement **LINE – STEP (10,10)**.

The advantage of relative positioning is that you don't have to identify the X,Y value of individual pixels. You need only know how far and in what direction you want to move.

Compare the following two programs. When you run them, you will find that the final results are the same. In the first program, however, eight absolute points have been identified—four starting points and four ending points.

```
10 SCREEN 4
20 CLS
30 LINE (10,10) - (10,30)
40 LINE (10,30) - (30,30)
50 LINE (30,30) - (30,10)
60 LINE (30,10) - (10,10)
70 END
```

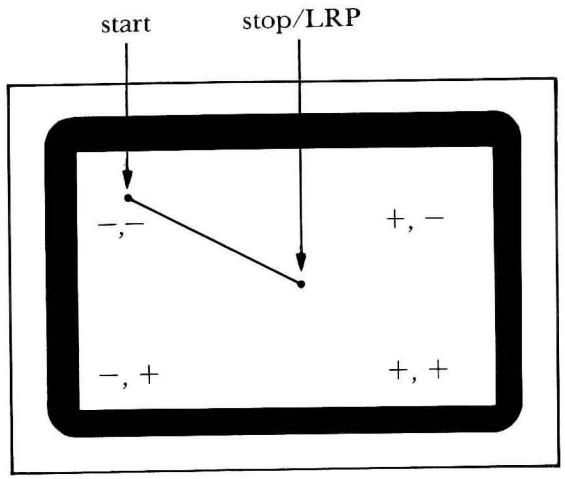


Figure 6-7. *Values indicating direction from last referenced point for relative moves from the LINE's ending point*

In the next program, we specify only a single absolute pixel with PSET.

```
10 SCREEN 4
20 CLS
30 PSET (10,10)
40 LINE - STEP (0,20)  'X stays the
   same, Y increases by 20
50 LINE - STEP (20,0)  'X increases
   by 20, Y stays the same
60 LINE - STEP (0,-20) 'X stays the
   same, Y decreases by 20
70 LINE - STEP (-20,0) 'X decreases by
   20, Y stays the same
80 END
```

The following program illustrates the STEP option's power. Five overlapping boxes, filled with different colors, are created with a single LINE statement that uses the STEP option. Furthermore, the only

absolute pixel is specified with PSET. As the program runs, you will be able to see the PSET STEP pixel (line 100) that identifies the last referenced point (LRP). (If you have trouble seeing the pixel, leave out the F option in line 90.) If this program didn't use relative positioning, it would be at least four times as long.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 X = 30  'X-axis diagonal value
50 Y = 20  'Y-axis diagonal value
60 C = 14  'Color starts at yellow
70 PSET (5,5)  'The only absolute point
80 FOR DELAY = 1 TO 600: NEXT DELAY
90 LINE - STEP (X,Y),C,BF
100 PSET STEP (-10,-10)   'This is
    the LRP and causes overlap
110 C = C - 1  'Color goes down from
    yellow
120 IF C < 1 THEN 140   'Change this
    value for fewer boxes
130 GOTO 80
140 END
```

The previous program points out how using relative positioning and variables makes large-scale changes to programs an easy matter. If you want a different number of squares, change line 120. If you want to start all of the squares at a different location, change line 70, and so on.

Line Clipping

At times your program may draw a figure that is too large for the PCjr display. When this happens, the parts of the figure that lap over the edges of the screen are clipped, and only the parts of the line or box that fit on the screen are displayed.

This means that some X,Y coordinates in your program may be outside the SCREEN's normal pixel range. For instance, SCREEN 5, with the greatest X value of 319 and Y value of 199, will allow you to locate points like X=400, Y=400. Y values greater than 200 scroll off the bottom of the screen, while those less than 0 (that is, negative numbers like −50) will scroll off the top of the screen. See Figure 6-8.

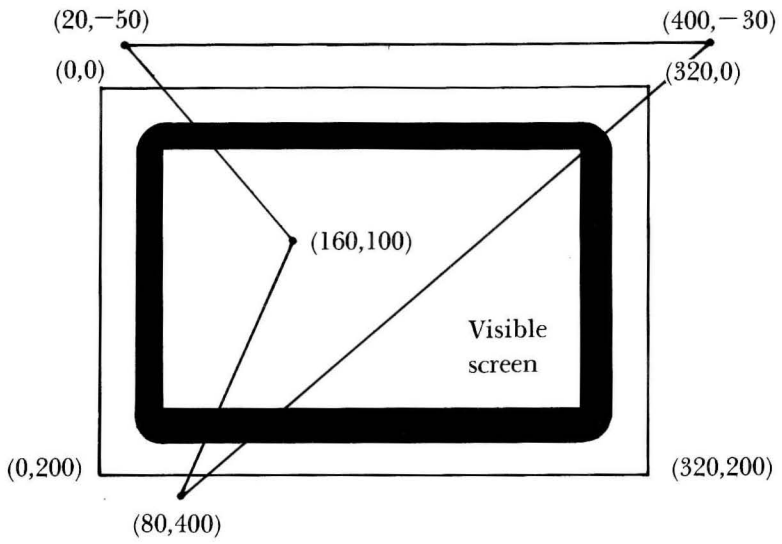


Figure 6-8. *Line clipping*

Nothing special is required in programs that specify values to be clipped. Simply indicate the X,Y locations as always. The following program lines produce figures that will be clipped.

```
40 LINE (100,40) - (80,400)
100 LINE (100,100) - (20,-40)
```

SAMPLE PROGRAM

One of the more practical applications of LINE is for making bar charts. The following program provides a simple bar chart with 12 vertical bars. All of the chart information is contained in the DATA statements in lines 210 and 220. If you want to modify the program for your data, change these lines. If you want more or fewer bars, change

lines 130 and 140, and if you want to change the grid marks in the background, change line 90.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 COLOR ,8
50 VIEW (20,20)-(250,150),,10
60 WINDOW (0,0)-(319,199)
70 X1 = 0: Y1 = 0: X2 = 320: Y2 = 0
80 LINE (X1,Y1) - (X2,Y2),7
90 Y1 = Y1 + 30: Y2 = Y2 + 30
100 IF Y1 > 200 THEN 120
110 GOTO 80
120 WINDOW (0,0) - (1,250)
130 FOR B = 0 TO 11    'B is the number
    of bars
140 W = 1/12    'W is the width of the
    bars
150 X = W * B
160 Y = 0
170 X2 = .8 * W
180 READ Y2
190 LINE (X,Y) - (X + X2, Y + Y2),10,BF
200 NEXT B
210 DATA 210, 165, 155, 125, 100, 75
220 DATA 35, 80, 40, 50, 70, 95
230 END
```

CHAPTER SEVEN

Drawing a Circle, Semicircle, Or Arc: CIRCLE

One of the graphics statements you will frequently use is CIRCLE. CIRCLE draws circles for pie charts, globes, balls, and other round figures. You should not have the impression, however, that this statement is only used to draw circles. It is one of the most versatile of all the PCjr graphics statements.

THE CIRCLE STATEMENT

In addition to complete circles, CIRCLE lets you set a pixel (like PSET) and create arcs, semicircles, and ellipses. Figure 7-1 illustrates various CIRCLE displays.

You have probably noticed that the more powerful and versatile a PCjr graphics command is, the more complicated it is. CIRCLE is no exception, as a review of Syntax 7-1 will show.

To draw a circle, you must specify the X,Y center point and the radius. Before reading this chapter, be sure that you are familiar with the PCjr's X,Y coordinate system, because this command, like most graphics statements, requires you to identify specific screen points on the display.

CIRCLE (*x,y*), *radius, color, start, end, ar*

CIRCLE STEP (*x,y*), *radius, color, start, end, ar*

x specifies the X-coordinate of the circle's center point. It is a numeric expression between
0 and 159 for SCREEN 3 (low resolution)
0 and 319 for SCREENs 1, 4, and 5 (medium resolution)
0 and 639 for SCREENs 2 and 6 (high resolution).
Values outside of these ranges will cause all or part of the circle to be displayed off the screen.

y specifies the Y-coordinate of the circle's center point. It is a numeric expression between 0 and 199 for all graphics display modes. Values outside of these ranges will cause all or part of the circle to be displayed off the screen.

radius specifies the circle's radius. It is a numeric expression between 0 and 32767.

color specifies the color of the circle (the foreground color). It is a numeric expression between
0 and 3 in SCREENs 1, 4, and 6
0 and 15 in SCREENs 3 and 5
0 or 1 in SCREEN 2.
color is optional; if omitted, the highest available color code is used.

start specifies the point where the figure is to begin. It is a numeric expression between +/−6.283185. *start* is optional; if omitted, +6.283185 is used.

end specifies the point where the figure is to end. It is a numeric expression between +/−6.283185. *end* is optional; if omitted, +6.283185 is used.

ar specifies the aspect ratio (major axis) of the figure. It is a numeric expression greater than 0. *ar* is optional; if omitted, 1.6 is used for low resolution, 0.83 is used for medium resolution, and 0.42 is used for high resolution.

STEP indicates the circle's center point is located relative to the last referenced point instead of an absolute screen location.

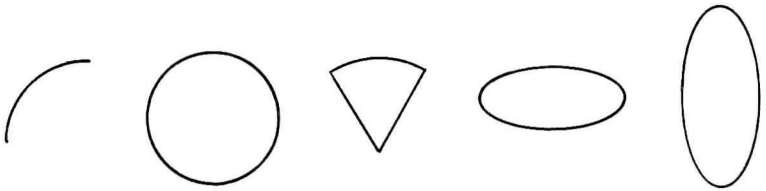


Figure 7-1. *Circle variations using the CIRCLE statement*

The Center Point

The first parameter used by CIRCLE defines the center point of the figure you wish to create. This center point is identified by a pair of numeric expressions representing the X-axis and Y-axis coordinates respectively. Even if you only want to draw a portion of a circle—an arc or semicircle—the center point must be specified, as Figure 7-2 illustrates.

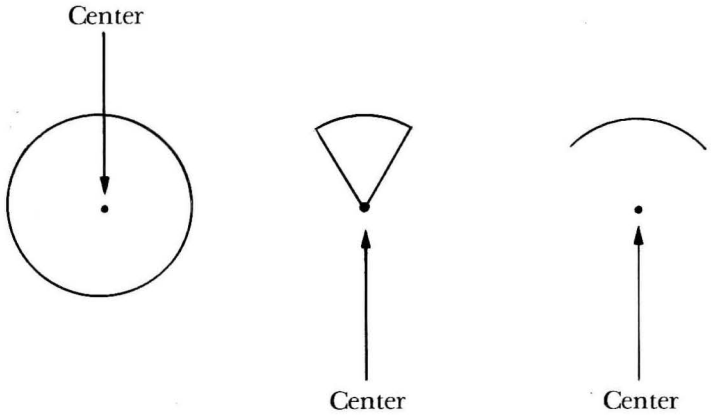


Figure 7-2. *The center of a circle*

Typically, the center point is specified in a program line like 50 CIRCLE (200,10). Or if you've assigned values to the variables X and y, a program might contain these lines:

```
50 X = 110
60 Y = 50
70 CIRCLE (X,Y)
```

The basic rules for identifying the center point are the following: the X,Y values must be inside parentheses, a comma must separate the two values, and the X value must always precede the Y value.

The X and Y values can be greater than or less than the allowable range for the resolution you're using. For instance, X can be greater than 159 if you're in low resolution, greater than 319 in medium resolution, and greater than 639 in high resolution. In all three resolution modes, Y can be greater than 199. X and Y can also be less than 0. However, if any of these conditions exists, part or all of the circle may not be visible on the screen.

The Radius

The second requirement for the completion of a circle is the *radius*—that is, the distance from the center point to the outer edge of the figure. Figure 7-3 illustrates the radius of a circle.

When you specify *radius*, you must be sure that it is preceded by a comma.

The radius cannot be greater than 32767, or an Overflow Error will occur. If the radius is greater than the visible screen coordinates, all or part of the circle may not be on the display. Figure 7-4 provides examples of legal and illegal radius values.

The following program creates a circle with its center point at the center of the screen and a radius of 50 pixels.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50
50 END
```

If your graphics program uses variables, you can use CIRCLE as

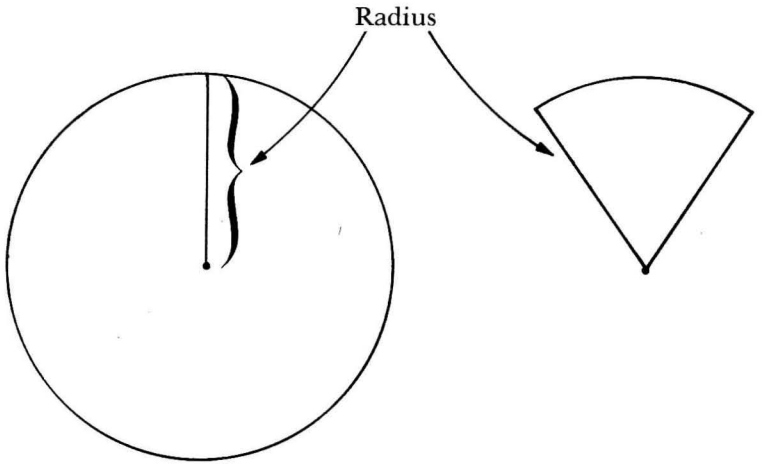


Figure 7-3. *Radius of a circle*

shown in the following program:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 X = 160
50 Y = 100
60 R = 50
70 CIRCLE (X,Y),R
80 END
```

CIRCLE (160,100),32767

Legal!

CIRCLE (160,100),32768
CIRCLE (100,100),−10

Illegal!

Figure 7-4. *Legal and illegal CIRCLE radius values*

The following program shows how a FOR-NEXT loop can create a series of ten concentric circles.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 FOR R = 1 TO 100 STEP 10
50  CIRCLE (160,100),R
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 NEXT R
80 END
```

For the most part, the radius is straightforward: you specify the length of the radius, and the PCjr draws the circle. As we mentioned in Chapter 2, however, pixels are not equilateral squares having equal heights and widths. In low resolution, the pixels are wider than they are tall, while in medium and high resolution, pixels are taller than they are wide.

When measurements on the X-axis and Y-axis are compared, radius lines are not precisely the same length. For example, in medium and high resolution, a distance of ten pixels up (on the Y-axis) is greater than a distance of ten pixels to the left or right (on the X-axis). Unless you specify otherwise, using the *ar* aspect ratio option discussed later in this chapter, the PCjr automatically makes adjustments so that circular objects are circular.

This discrepancy between a pixel's height and width affects all graphics. However, its effect is more noticeable and can cause more problems with CIRCLE than with any other statement. If you need to exert more control over the pixel's height/width ratio, use the *ar* option.

The Color

Once you've identified a circle's two required parameters—its center and size—you can begin manipulating the available options to further design your figure. The first of these options is the color.

When you use the color option, you are actually coloring the circle's border, as well as any radius lines generated by the CIRCLE statement. See Figure 7-5.

The color option for CIRCLE is much like the color option for LINE; the color you get depends upon the SCREEN you're using. Since *color* is optional, the PCjr uses the highest available color code

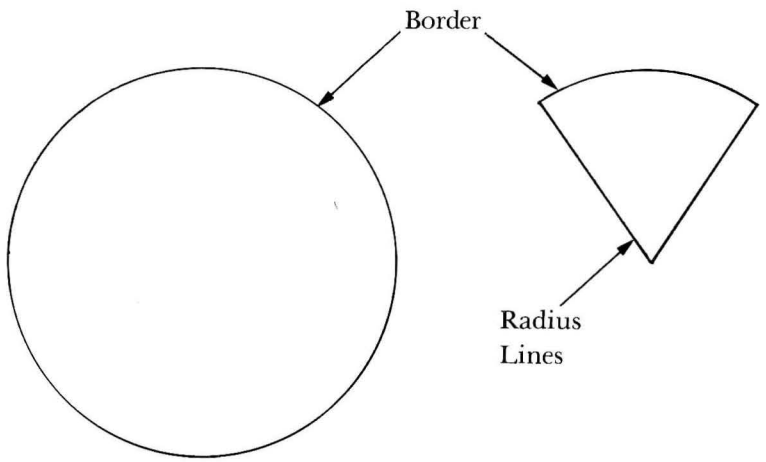


Figure 7-5. *Areas affected by CIRCLE's color option*

when you don't specify a color. In SCREENs 4 and 6, color code 3 will be used; in SCREENs 3 and 5, 15 will be used. However, you can use any other color by including the color option. For instance, the following short program will draw a white circle (color code 15) on a black background:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (100,100),50
50 END
```

Since this program uses SCREEN 5 (a 16-color mode), you can select any of the available colors simply by adding the color option in program line 40. Editing the program as follows causes a yellow circle to be drawn on a black background:

```
40 CIRCLE (100,100),50,14
```

However, in SCREEN 1, 4, or 6, the only foreground colors available when you enter Graphics Mode are cyan, magenta, and white (position

codes 1, 2, and 3). The next program will draw a high resolution, magenta circle on a black background.

```
10 CLEAR,,,32768
20 SCREEN 6
30 CLS
40 CIRCLE (100,100),50,2
50 END
```

If you're using a 4-color mode and you want to use a color other than cyan, magenta, or white, you'll need to use PALETTE to reassign color codes 1 through 3.

Since the value of the color option can be a numeric expression, you can also use it in this way:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 FOR C = 1 TO 15
50  CIRCLE (160,100),50,C
60 FOR DELAY = 1 TO 100: NEXT DELAY
70 NEXT C
80 END
```

In this example, a 16-color mode is selected and the background is set to black. Then a circle is drawn in the center of the screen. After a few seconds, the circle begins cycling through the available colors.

The color option gains particular importance when you consider that CIRCLE, unlike LINE, does not have an automatic "fill" option. If you want to fill a circle with color, you must use the PAINT command. In most cases when you use PAINT, you'll need to specify the color of the border at which to quit painting. In this way CIRCLE's color option lets you control which areas are to be painted and which areas are not. In the following program, for example, two concentric circles are drawn, a blue circle and a yellow one. Next, the PAINT command paints with red until a yellow border is encountered. Notice that the green circle is painted over.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50,1
50 CIRCLE (160,100),100,14
60 PAINT (160,100),4,14
70 END
```


Drawing Arcs or Semicircles

In addition to complete circles, CIRCLE generates arcs, semicircles, and "pieces of a pie" for pie charts. CIRCLE's *start* and *end* options let you do this.

To use the start and end point options, however, you must think in terms of angles, *radians* (a unit of angular measurement), and degrees as units of measurement instead of pixels. If you are familiar with geometric figures like circles, you've heard of pi (the ratio of a circle's circumference to its diameter), which is equal to 3.1415926535 radians. A complete circle is 360 degrees, which equals 6.283185 radians.

How can you easily measure radians on a PCjr circle? Think of a circle in terms of a clock as shown in Figure 7-6.

A full circle is equal to 2 pi and the PCjr assigns this value to a point that is equivalent to 3:00. The PCjr always draws the circle counterclockwise starting at 3:00. 12:00 is 1/2 pi, 9:00 is equal to 1 pi, and 6:00 is 1 1/2 pi. When 3:00 is again reached, it is interpreted as 2 pi. Figure 7-7 shows the clock and radian equivalences.

If you are used to measuring circles and angles in degrees instead of radians, you can convert radians to degrees. The circle always starts at

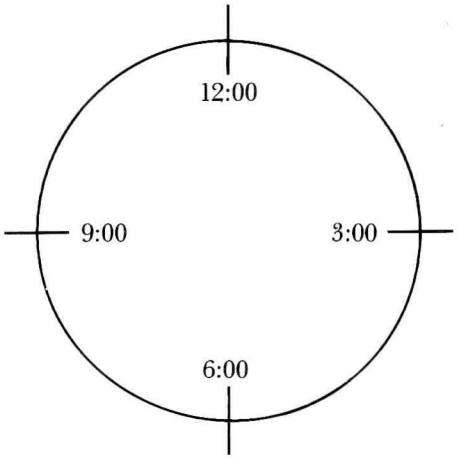


Figure 7-6. *Circle with clock equivalences*

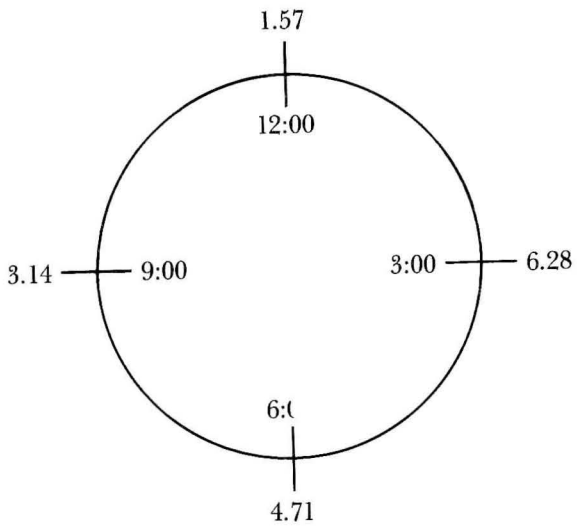


Figure 7-7. *CIRCLE's clock and radian equivalences*

3:00 and continues counterclockwise; 3:00 is 0 degrees, 12:00 (or 1/2 pi) is 90 degrees, 9:00 is 180 degrees, and so on. Table 7-1 and Figure 7-8 provide a complete description of degree, radian, and clock equivalences.

If you want to draw a 90-degree arc, for example, you might say

Table 7-1. *Circle Measurement Equivalences*

Clock	Radians	Degrees	pi
3:00	0	0	0.0
12:00	1.57	90	0.5
9:00	3.14	180	1.0
6:00	4.71	270	1.5
3:00	6.28	360	2.0

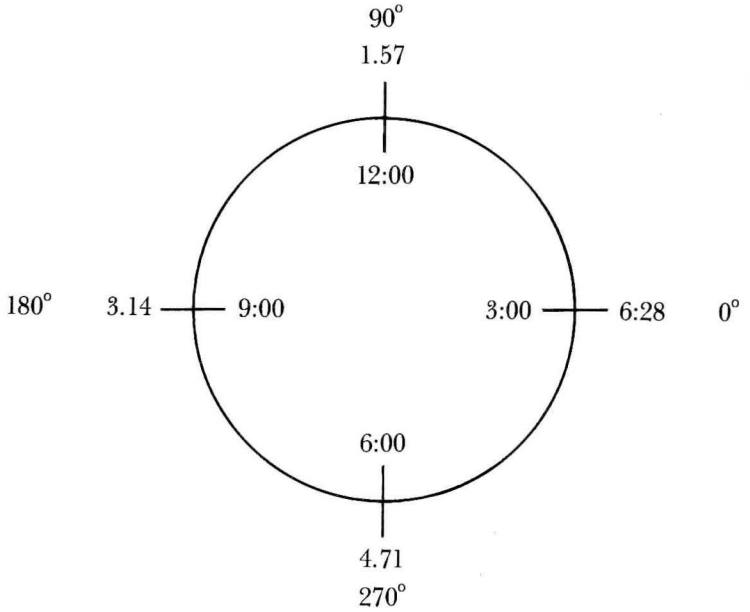


Figure 7-8. *Circle equivalences*

(after translating into BASIC) "Start drawing at 3:00, go counterclockwise, and stop drawing at 12:00." This would produce an arc similar to the one shown in Figure 7-9.

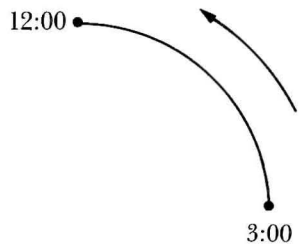


Figure 7-9. *Sample 90-degree arc*

Notice that you can't simply reverse the order of the start and end points and have the same results. For instance, if you said (in BASIC), "Start drawing at 12:00 and stop drawing at 3:00," you would create a 270-degree arc (see Figure 7-10), since circle drawing is counterclockwise.

Whatever terms you use to measure a circle's measurements (degrees, radians, or hours on the clock), the PCjr always thinks in terms of radians. Therefore, you must specify a radian equivalence (between −6.283185 and +6.283185) when using CIRCLE's start point and end point options, as in this program:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50,,1.57,3.14
50 END
```

Here, an arc (with the center in the center of the screen) is drawn counterclockwise from 12:00 to 9:00. Figure 7-11 illustrates this concept.

In the last example, the start and end points were unequal, positive values. However, you can enter negative values (except for −0) and a

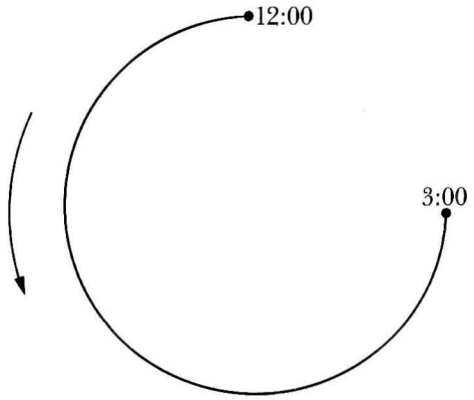


Figure 7-10. *Sample 270-degree arc*

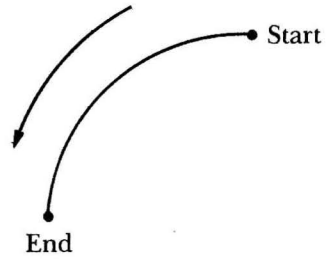


Figure 7-11. *Sample CIRCLE statement with positive start and end points*

radius line will be drawn from the circle's circumference to the center point. This is how you can create a "piece of a pie." If you change line 40 of the previous example, a pie section like that in Figure 7-12 will be drawn.

```
40 CIRCLE (160,100),50,,-1.57,-3.14
```

If the start and end points are equal to each other, a circle will not be drawn; instead a pixel will be set (turned on)—just as if you had used

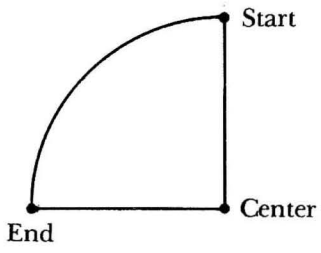


Figure 7-12. *Sample CIRCLE statement with negative start and end points*

PSET. And as in other PCjr graphics statements, you can use variables to represent start and end point values.

The following program shows how arcs with radius lines can be used with CIRCLE. Line 40 draws a yellow circle with a "piece of pie" missing; line 50 draws the green "piece of pie" that has been set aside, as illustrated in Figure 7-13. If the start and end options were positive instead of negative values, the arcs would have been drawn without the lines.

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50,14,-3.14,-1.57
50 CIRCLE (150,90),50,2,-1.57,-3.14
60 END
```

The next program, adapted from *Graphics Primer for the IBM PC* (Mitchell Waite and Christopher Morgan, Osborne/McGraw-Hill, 1983), is another example of how CIRCLE can be used to create arcs.

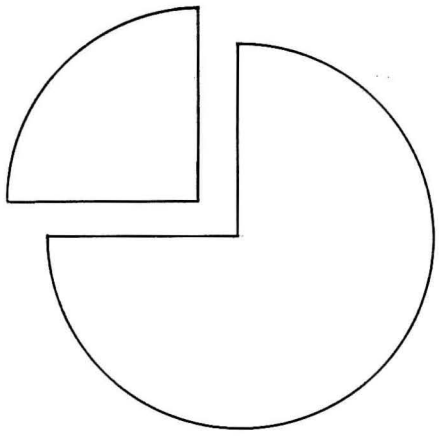


Figure 7-13. *Negative start and stop points used to draw radius lines on arcs*

Each time an arc is drawn, the center point is recalculated and successive arcs are longer than previous ones until a full circle is created.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 PI = 3.1415926535
50 R = 40
60 FOR E0 = 0 TO 2 * PI + .01 STEP PI/5
70 Y = 100 - R * SIN(E0)
80 X = 160 + R * COS(E0)
90 CIRCLE (X,Y),R,14,0,E0
100 NEXT E0
110 END
```

The Aspect (Height/Width) Ratio

The last option that CIRCLE provides is *ar*, the aspect ratio (sometimes referred to as the height/width ratio). The aspect ratio lets you create ellipses like the ones shown in Figure 7-14.

If each pixel on the screen were an exact square, you wouldn't need to worry about the ratio of the height to the width. As we pointed out earlier, however, pixels in low-resolution mode are wider than they are

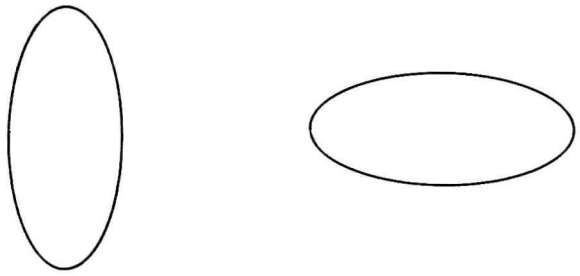


Figure 7-14. *Ellipses created with CIRCLE's* ar *option*

tall, while pixels in medium- and high-resolution modes are taller than they are wide. In the medium- and high-resolution modes, therefore, a line of ten pixels on the X-axis is actually longer than a line of ten pixels on the Y-axis, as Figure 7-15 illustrates.

Normally, the PCjr compensates for these differences. The computer uses the default *ar* values of 1.6 for low resolution, 0.83 for medium resolution, and 0.42 for high resolution. The result of these adjustments is that circles are actually round. If the adjustments weren't made, and all circle measurements were on a 1-for-1 basis, circles would be egg-shaped.

To illustrate this concept, type and run the next program, which begins by drawing a circle that does not use the *ar* option. After a short delay, a second circle is drawn (line 60) with the *ar* equal to the default value (the circle is still round). In line 80, the aspect ratio changes to 1, which creates an ellipse that is taller than it is wide. Finally, in line 100

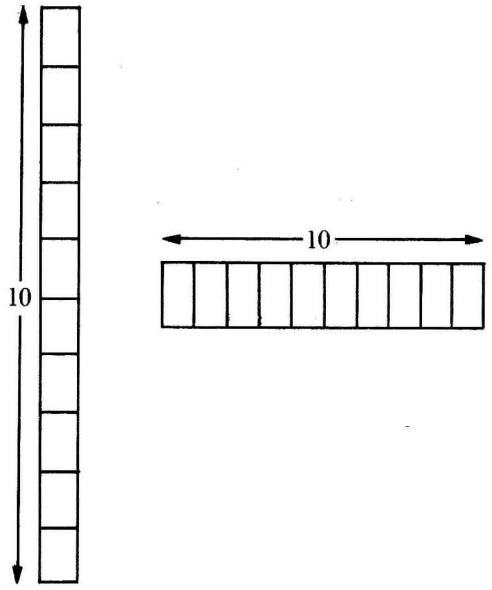


Figure 7-15. *Relative difference in X-Y radius*

just the opposite is drawn when the aspect ratio is less than the default value.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50  'Default ar =
   .83
50 FOR DELAY = 1 TO 600: NEXT DELAY
60 CIRCLE (160,100),50,,,,.83  'No
   change in this circle
70 FOR DELAY = 1 TO 600: NEXT DELAY
80 CIRCLE (160,100),50,,,,1  'Taller
   than wide ellipse
90 FOR DELAY = 1 TO 600: NEXT DELAY
100 CIRCLE (160,100),50,,,,.5  'Wider
    than tall ellipse
110 END
```

If the height equaled the width for all pixels, an aspect ratio of 1 would create a round circle, while a greater-than or lesser-than value would create an ellipse. As you can see, this is not the case.

In mathematical terms, the relationship of a pixel's height to its width is described like this:

$$ar = \frac{\textit{length of Y-axis}}{\textit{length of X-axis}}$$

If all pixels were equilateral squares, 1 would be equal to 1/1. But again, that's not the case with PCjr pixels. Low-resolution pixels use the height/width ratio of 5/3, medium-resolution pixels use 5/6, and high-resolution pixels use 5/12 to create a circle. If you apply these ratios to the above formula, the results are 1.6 for low resolution, 0.83 for medium resolution, and 0.41 for high. This means that when these values (the default values as it happens) are used for the aspect ratio values, a round circle is drawn.

Remember that if *ar* is less than the default, the radius is measured on the X-axis and the ellipse is taller than it is wide. If *ar* is greater than the default, the radius is measured on the Y-axis and the ellipse is wider than it is tall. This is why lines 80 and 100 in the previous program created the ellipses they did. Figure 7-16 also illustrates these differences.

Since the aspect ratio can be expressed as a variable, the following

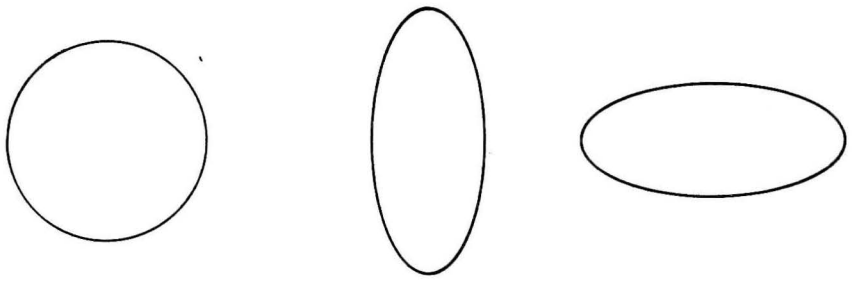


Figure 7-16. *Effect of aspect ratio on CIRCLE*

program will let you watch how an ellipse can increase in size:

```
10 CLEAR,,,32768
20 SCREEN 5
30 CLS
40 FOR AR = 0 TO 5 STEP .25
50 CIRCLE (160,100),60,,,,AR
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 NEXT AR
80 END
```

Because the aspect ratio is equal to 0, the first figure you will see when you run this program is a straight line. The figure then gradually grows into a "standing" ellipse ($ar < 0.83$) until a full circle is reached ($ar = 0$); then the figure begins growing into a Y-axis ellipse ($ar > 0.83$).

The next program illustrates both types of ellipses, as well as circles. When you run the program, 25 "lying down" ellipses are drawn, then 25 "standing up" ellipses, and finally, 25 regular circles.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 COLOR ,7
50 FOR R1 = 10 TO 250 STEP 10
60 CIRCLE (160,100),R1,4,,,,.5
```



```
70 NEXT R1
80 FOR R2 = 10 TO 250 STEP 10
90 CIRCLE (160,100),R2,1,,,,1.5
100 NEXT R2
110 FOR R3 = 10 TO 250 STEP 10
120  CIRCLE (160,100),R3
130 NEXT R3
140 END
```

Relative Moves and the STEP Option

The one option we haven't discussed yet, STEP, uses relative positioning to identify a circle's center point. Remember, there are two ways to specify X,Y coordinates for LINE. One way is the *absolute* method, which is the one we've been describing so far. The other method is the *relative* method in which you indicate a coordinate that is "so many pixels away from the current or last referenced pixel."

When using the STEP option's relative positioning capability, you must think of locations that are greater than or less than the last referenced or current point. Figure 7-17 illustrates this concept.

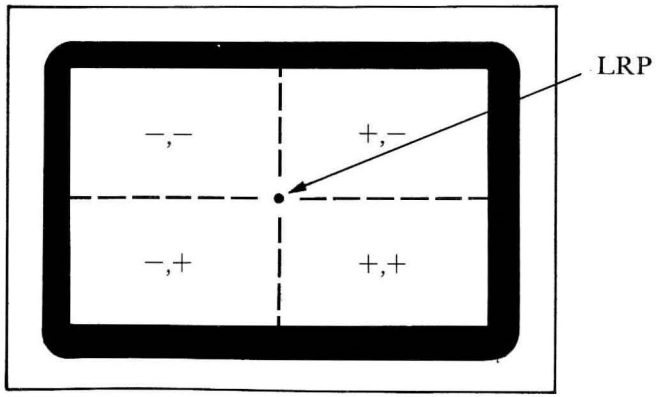


Figure 7-17. *Values indicating direction from last referenced point for relative moves from the CIRCLE's center point*

Suppose you drew a circle in the middle of your screen by using **CIRCLE (160,100),30**. Now you want to draw another circle 50 pixels to the right on the X-axis and 50 more spaces down on the Y-axis, but you don't know that particular absolute coordinate position. In this case you would want to use a relative move, and you would only need to enter the statement **CIRCLE STEP (50,50),20**.

The advantage of relative positioning is that you don't have to identify the X,Y value of individual pixels. You only have to know how far and in what direction you want to move.

The following program illustrates how CIRCLE can use the STEP option to create a graph that uses overlapping circles to show relationships. This popular graph is often used for financial planning to show overlap between departments or expenses. In our simple example, the dark blue areas are areas unaffected by other circles, the light blue is where two circles overlap, and the red area is where all three circles touch.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),40    'Absolute point
50 CIRCLE STEP (30,20),40 'Relative move
60 PRESET (160,100) 'Resets LRP
70 CIRCLE STEP (-30,20),40 'Relative move
80 PAINT (160,90),1,15
90 PAINT (120,120),1,15
100 PAINT (180,100),1,15
110 PAINT (150,100),9,15
120 PAINT (180,100),9,15
130 PAINT (160,120),4,15
140 PAINT (160,140),9,15
150 END
```

SAMPLE PROGRAM

The final program in this chapter is again adapted from Waite and Morgan's *Graphics Primer for the IBM PC*. The object of this program is to create the skeleton of a globe. Most of the graphics commands we have introduced are used in this program.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 COLOR 1,7
```



```
50 XO = X: XO = XO + RD * .1: YO = 0:
   RD = 15: RD = RD * .9
60 DIM X(20,20), Y(20,20)
70 'Grid constants
80 PI = 3.14159
90 R = 12        'radius of globe
100 A = .7       'tip angle of globe
110 COSA = COS(A): SINA = SIN(A)
120 'Grid values
130 FOR I = 0 TO 10
140 T = I * PI / 10
150 COST = COS(T): SINT = SIN(T)
160 FOR J = 0 TO 10
170 S = J * PI / 10
180 COSS = COS(S): SINS = SIN(S)
190 X(I,J) = R * COST * SINS
200 Y(I,J) = R * SINA * SINT * SINS + R
    * COSA * COSS
210 NEXT J
220 NEXT I
230 VIEW (60,20) - (260,180),,3
240 CLS
250 WINDOW (XO-RD, YO-RD) - (XO + RD,
    YO + RD)
260 ' Latitudes
270 ASPO = .67 * COSA
280 FOR S = 0 TO PI STEP PI / 10
290 X = 0
300 Y = R * COSA * COS(S)
310 RO = R * SIN(S)
320 CIRCLE (X,Y),RO,1,,,ASPO
330 NEXT S
340 'Longitudes
350 FOR I = 0 TO 20
360 PSET (X(I,0),Y(I,0))
370 FOR J = 1 TO 10
380 LINE -(X(I,J),Y(I,J))
390 NEXT J
400 NEXT I
410 END
```

CHAPTER EIGHT

Drawing "Freehand": DRAW

DRAW is the most powerful, versatile, and ultimately the most flexible of all the PCjr graphics commands. Unlike the other graphics statements we've introduced, you can use DRAW almost as you would a pencil or pen.

One feature that makes DRAW so powerful is its set of subcommands, generally referred to as a *graphics definition language (GDL).* This subset provides a unique version of each primary PCjr BASIC graphics command, including X,Y moves (absolute and relative), color control, and painting. The GDL also has some features not directly available with the primary commands. These features include scaling and angling.

Remember that *DRAW uses string expressions, instead of numeric expressions, to create objects.* Consequently, DRAW's parameters and options must be enclosed in quotation marks if they are string constants. If parameters and options are assigned to string variables, the variable names must end with the dollar sign suffix. This is different from the PCjr graphics statements you learned earlier.

With DRAW, all screen activity starts at the *last referenced point (LRP).* You should think of the LRP as a pencil point that you move around the screen. The first thing you must do is tell the PCjr where to begin drawing. The PCjr will set the LRP to that point. Next you must specify the direction in which you want to draw. Then indicate how far drawing is to continue in that direction. Those are the basic

Table 8-1. *DRAW Movement Commands*

Movement Commands	
Parameter	**Description**
M*x,y*	Moves the draw position to a specified X,Y position. *x* is a numeric expression between 0 and 159 for low resolution, 0 and 319 for medium resolution, 0 and 639 for high resolution. In all resolutions, *y* is between 0 and 199. **M** may be an "absolute move" or a "relative move." If *x* and *y* are prefaced by + or − signs, the move will be relative.
U *number*	Moves *up* a specified number of pixels.
D *number*	Moves *down* a specified number of pixels.
L *number*	Moves *left* a specified number of pixels.
R *number*	Moves *right* a specified number of pixels.
E *number*	Moves at a 45-degree angle for a specified number of pixels (diagonally, up, and to the right).
F *number*	Moves at a 135-degree angle for a specified number of pixels (diagonally, down, and to the right).
G *number*	Moves at a 225-degree angle for a specified number of pixels (diagonally, down, and to the left).
H *number*	Moves at a 315-degree angle for a specified number of pixels (diagonally, up, and to the left).
B	Specifies that the subsequent lines are "blank" lines.
N	Specifies that the subsequent lines are "no update" lines.

steps, although using DRAW becomes more complex when you create invisible lines, change line colors, modify the scale or angle, and so on. Consequently, we suggest you take extra time to learn DRAW, probably more time than you spent learning other PCjr graphics statements. Also, be sure you are familiar with the screen X,Y coordinate system.

THE DRAW STATEMENT

DRAW's syntax (shown in Syntax 8-1) is deceptively simple because it has only a single parameter. However, that parameter can take a number of different forms. Tables 8-1, 8-2, and 8-3 describe the subcommands used with the DRAW statement.

DRAW *object*

object is a string expression and may include movement commands, modes commands, and command prefixes. Tables 8-1 through 8-3 describe these parameters.

Note: If *object* is a string constant, it must be enclosed in quotation marks. If *object* is a string variable, it must end with a dollar sign ($).

Syntax 8-1. *The DRAW Statement*

Identifying and Moving the Drawing Position

DRAW initially requires you to specify where drawing is to begin. You identify this point by entering the M command followed by an X,Y position (see Table 8-1). M instructs the PCjr to set the LRP to the location identified by *x,y* and wait for the next instruction. (M is optional; if it is omitted, the PCjr uses the center of the screen as the LRP.) Once identified, this position becomes the LRP, and subsequent drawing takes place in relation to it, unless you specify otherwise. In a pro-

Table 8-2. *DRAW Mode Commands*

Mode Commands	
Parameter	**Description**
C *number*	Specifies the color of an available foreground color. *number* is a numeric expression between 0 and 3 for SCREENs 1, 4, and 6 1 and 15 for SCREENs 3 and 5 0 or 1 for SCREEN 2. When this option is not used, the current foreground color (default value) is used.
A *number*	Turns the drawing direction at the LRP to the specified angle. For all resolutions, *number* is a numeric expression between 0 and 3: 0 = 0 degrees 2 = 180 degrees 1 = 90 degrees 3 = 270 degrees. When A is not used, A0 is the default value.
TA *angle*	Turns the drawing direction at the LRP to the specified angle. For all resolutions, *angle* is a numeric expression between −360 and +360 degrees.
S *number*	Sets the scale factor to make objects larger or smaller. For all resolutions, *number* is a numeric expression between 1 and 255. When S is not specified, S4 is used as the default value.

gram, DRAW is used like this:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "M160,100"
```

When you run this program, all you see is a dot in the center of the screen. This pixel is the first point at which drawing will begin and is located at X=160, Y=100.

Table 8-3. *DRAW Command Prefixes*

Command Prefixes	
Parameter	**Description**
X	Specifies that the next command is a substring to be executed.
P *paint,border*	Fills or "paints" an enclosed area with the specified foreground color (*paint*), stopping at the specified *border* color.

Identifying an X,Y location like this is called "absolute positioning" because drawing begins at a specific, absolute screen position. M lets you move the LRP anywhere within the maximum screen limits, including locations off the visible screen. Change line 40 in the previous program to the following line and run the program.

```
40 DRAW "M160,100 M319,199"
```

A line (from the center of the screen to the lower right corner) appears because anytime the LRP is moved, a line is drawn from the old position to the new position. It is as if a drawing pencil were moved from point to point, connecting the dots with straight lines.

Drawing a Blank Line

It is possible to tell the computer to move the LRP without drawing a visible line in the process (an equivalent of picking up the drawing pencil). To do so, use the B (blank line) option described in Table 8-1. Any move immediately following the B option will move the LRP without leaving a trace. Change line 40 in the last program to the following line:

```
40 DRAW "M160,100 BM319,199"
```

When you run the program this time, you will see a dot in the center of the screen again. But something else has happened. Add the follow-

ing line to see what we mean:

```
50 DRAW "M319,0"
```

Now when you run the program, a line is drawn from the lower right corner to the upper right corner of the screen, as illustrated in Figure 8-1.

If you delete the B option from line 40, a solid line is drawn from the center of the screen to the lower right corner, then to the upper right corner.

Returning the Drawing Position to a Previous Position

The B option isn't the only choice you have when drawing lines, however; DRAW also has the N (no update) line (refer to Table 8-1).

Normally when you use DRAW to draw a line, you start at point A and draw to point B. The drawing sequence after that starts at point B.

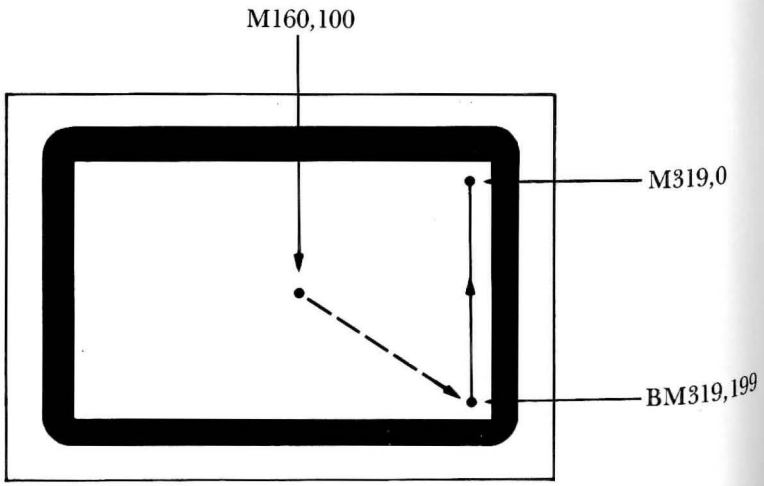


Figure 8-1. *Effect of visible and invisible moves*

The N option automatically returns the LRP to the original starting point (point A) before drawing the next line. In other words, there is "no update" to the LRP. See Figure 8-2.

To draw a line without updating the LRP, simply enter N directly before the move or line that is not to be updated. The following program draws perpendicular lines in the center of the screen. At each move, the LRP returns to the center of the screen before drawing subsequent lines. If you delete even one N from line 40, the figure will be much different.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "M160,100 NM160,50 NM210,100 NM1
   60,150 NM110,100"
50 END
```

N can be used with the B option on as well.

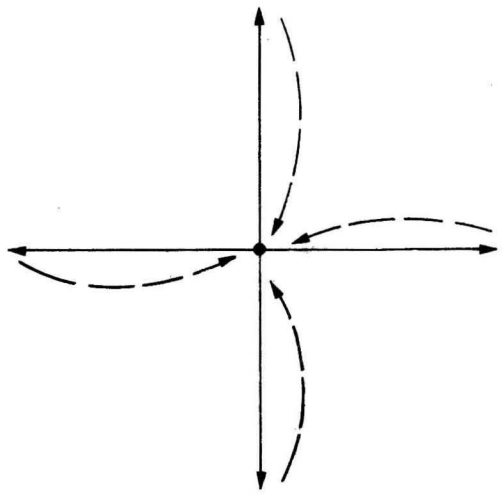


Figure 8-2. *DRAW's N option does not update the LRP*

Giving Direction to the DRAW Position

Once you've identified where the drawing will start, you must specify where it will go (see Table 8-1). There are eight pre-set directions, each 45 degrees from the other and each indicated by a specific subcommand, as Figure 8-3 illustrates.

To use a directional command, simply type in the letter that specifies the direction you want a line drawn. For example, the E in line 40 tells the PCjr to begin drawing a line 45 degrees (up and to the right) from the LRP.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "M160,100 E
```

It is not enough simply to give the direction, however; you must also tell the PCjr how far to draw the line. (If you use a directional command and don't say how far to go, the PCjr will move the LRP a distance of one pixel, as it did in the previous program.) The PCjr will let you move the LRP from −99999 to +99999, although only those locations within the visible boundaries of the current screen resolution will be seen. This means that X can be between 0 and 159 for SCREEN 3, between 0 and 319 for SCREENs 1, 4, and 5, and between 0 and 639 for SCREENs 2 and 6. Y is between 0 and 199 for all resolution modes. For example, the following program will draw a line from the center of the screen, up and to the right 50 pixels.

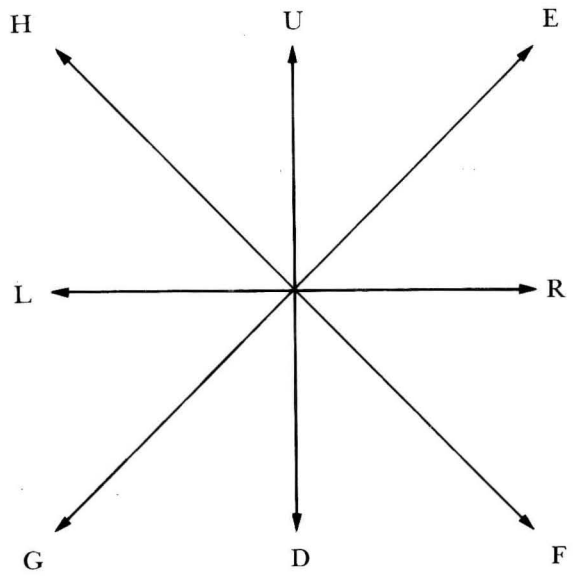


Figure 8-3. *DRAW's directional commands*

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "M160,100 E50
```

The following program draws a square with the lower left corner at the center of the screen. (Refer again to Figure 8-3 for a description of directional subcommands.) Notice how the R (right), U (up), L (left), and D (down) subcommands are used to indicate the "next" direction, while 50 pixels is the length in every case.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "M160,100 R50 U50 L50 D50"
50 END
```

The other four directional subcommands can be used also. In this example, the *lower* tip of a diamond is at the center of the screen.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "M160,100 E50 H50 G50 F50"
50 END
```

Absolute Moves and Relative Moves

In the previous examples, all moves other than the initial (absolute) moves have specified the direction and length of the move. They have not been drawn to a specific screen location. **R50**, for example, isn't a

screen location; it is simply an instruction to move 50 pixels to the right. There is no reference to a specific X,Y location. The next program draws the same square, but uses absolute moves (meaning that specific X,Y locations are identified) in every instance. In this example, there is no explicit instruction for direction or length.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "BM160,100 M210,100 M210,50
   M160,50 M160,100"
50 END
```

Another type of move, a relative move, can also be made. This move is similar to the STEP option used with LINE, CIRCLE, PSET, and so on. Instead of using **STEP**, you preface each X,Y identifier with a plus or minus sign. If the X value is prefaced with a +, the value will be added to the current X value; if X is prefaced by a −, the indicated number of pixels will be subtracted from the current X value. The same holds true for Y. The absence of a sign indicates an absolute move.

To understand why you might want to use this version of M, consider this situation. Suppose you have drawn a box beginning at a certain screen location. Then suppose you want to draw another box at a location relative to the first square, but you don't want to have to define all those points again (and an absolute move would require you to do this). A "relative move" is the easiest way to accomplish this. Relative screen coordinates are specified in relation to the last referenced point (for instance, 100 DRAW "M+40,−50"). The following program draws another box, this time using relative moves. Again, there are no explicit instructions for direction or length.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DRAW "BM160,100 M+50,+0 M+0,-50 M-50,
   +0 M+0,+50"
50 END
```

We've introduced three ways DRAW can create the same object. It should be obvious that the first method, specifying direction and length, is much more efficient, at least in the situations we have used.

Punctuation and Variables

As you have seen throughout this book, X,Y values *must* be separated by a comma. This is also true with DRAW. However, a blank space or a semicolon can be used to separate subcommands as well as to indicate the end of a command.

Blanks and semicolons are used to improve the readability of a program line. In long programs, however, they can consume valuable memory space. Consequently, the method you use to differentiate between actions is up to you. In this chapter we use spaces where appropriate, since it makes reading the programs easier. When writing your own programs, especially if they are long, you may want to delete spaces within the command line. All of the following lines are valid:

```
40 DRAW "M160,100 R50 U50 L50 D50"
40 DRAW "M160,100;R50;U50;L50;D50;"
40 DRAW "M160,100R50U50L50D50"
```

Nevertheless, there are times when the semicolon is required. One of these conditions is when variables are used with a command. Usually you are assigning a value to a variable, and then using that variable in a program line. For instance, X=100; Y=100; R=50; CIRCLE (X,Y),R.

Because of the variety of DRAW subcommands, simple use of variables just isn't possible. Therefore, DRAW requires that once a value is assigned to a variable, that variable *must* be prefaced by an equal sign.

Furthermore, when you use variables and equal signs in a statement, DRAW also *requires* that you separate directional commands with a semicolon. An Illegal Function Call error will result if you don't. See Figure 8-4.

The next program, for example, draws the same square we have been creating, but uses variables to do so.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 A = 50
50 DRAW "M160,100 R=A;U=A;L=A;D=A;"
60 END
```

As you know, the use of variables and relative moves can make programs simpler to write, but powerful nonetheless. The next example demonstrates a simple way to create a cube of apparently three dimen-

```
                              50 A = 50
                              60 DRAW "UA DA"
50 A = 50                     50 A = 50
60 DRAW "U=A;D=A;"            60 DRAW "U=A D=A"
                              50 A = 50
                              60 DRAW "UA;DA"
```

Right! *Wrong!*

Figure 8-4. *Legal and illegal use of DRAW variables and delimiters*

sions using variables and relative moves. Lines 50 and 60 draw the two cube faces, while line 70 connects the corners. See Figure 8-5.

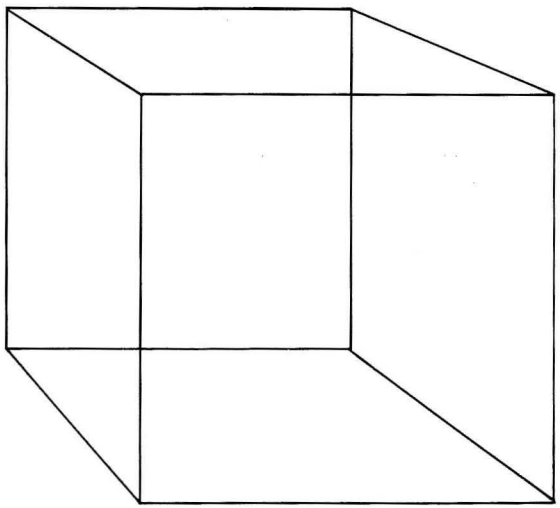


Figure 8-5. *Three-dimensional cube created by DRAW's relative moves and variables*


```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 A = 50
50 DRAW "M160,100 U=A; L=A; D=A; R=A;"
60 DRAW "BM+20,+20 U=A; L=A; D=A; R=A;"
70 DRAW "M160,100 U=A; M+20,+20 L=A;
   M-2 0,-20 D=A; M+20,+20"
80 END
```

Another way of using variables with DRAW is to enter a *draw string* in which the entire instruction set used to create an object is assigned to a string variable. The DRAW statement then executes the string variable. This increases a program's flexibility by allowing you to redraw the same object over and over while entering the necessary command sequence only once.

This example shows how the cube we created in the previous example can be drawn with string variables. Notice that the instructions for creating the box are stored in line 45. That string is then executed in lines 55 and 65. This has two useful features: it spares you from having to type the same thing over and over, and it allows you to modify a single instruction but have the change affect the entire program.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 A = 50
45 BOX$ = "U=A; L=A; D=A; R=A;"
50 PSET (160,100)      'Sets LRP
55 DRAW BOX$           'Executes string
60 DRAW "BM+20,+20"    'Relative move
65 DRAW BOX$           'Executes string
                        again
70 DRAW "M160,100 U=A; M+20,+20 L=A; M-2
   0,-20 D=A; M+20,+20"
80 END
```

Storing the instructions that create a graphics object in string variables is extremely useful for techniques like animation. You can, for instance, enter once the instructions that draw a boat (like the one shown) and move the boat across the screen by executing the draw string over and over.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 BOAT$ = "R60 U40 G30 R25 D10 R30 G20
   L45 H20"
50 PSET (X,100)
60 DRAW BOAT$
70 X = X + 30
80 IF X > 200 THEN END
90 CLS
100 GOTO 50
110 END
```

Admittedly, this isn't the best animation technique, but it does illustrate the fact that DRAW is a very powerful statement.

Controlling the Color and Filling an Area

DRAW also lets you determine the color of an object (see Table 8-2) you are drawing—depending, of course, on the available foreground colors. As with other graphics statements, the highest available color code will be used for the foreground color unless you specify otherwise. The number of colors available depends upon the resolution mode you are using. If you are using a 4-color mode and want colors other than the default ones, you will have to use COLOR or PALETTE.

The C (color) subcommand lets you specify or change the current foreground color. To use C, simply enter the command with a suffix indicating the color code of the desired color—for instance—100 DRAW "C4 U50 R50 D50 L50". You can assign the color code to a numeric variable. However, DRAW requires that you follow two rules when you execute the variable: an equals sign *must* precede the variable, and a semicolon *must* follow the variable. See Figure 8-6.

For example, if you wanted the boat we created earlier to be red, both of the following program modifications would be valid:

```
40 BOAT$ = "C4 R60 U40 G30 R25 D10 R30 G
   20 L45 H20"
```

```
35 C = 4
40 BOAT$ = "C=C; R60 U40 G30 R25 D10 R30
   G20 L45 H20"
```


	40 K = 4
50 DRAW "C4"	50 DRAW "CK"
40 K = 4	40 K = 4
50 DRAW "C=K;"	50 DRAW "C=K"
	40 K = 4
	50 DRAW "CK;"
Right!	*Wrong!*

Figure 8-6. *Legal and illegall methods of using variables with*

You can change the color anywhere in an instruction set, and that change will affect all of the drawing actions until you specify otherwise. In the boat example, for instance, you can make the hull red and the sail blue by changing line 40 to the following:

```
40 BOAT$ = "C4; R60 U40 C1; G30 R25 D10
   C4; R30 G20 L45 H20"
```

C4 sets the color to red, C1 sets it to blue, and C4 sets it back to red. (The semicolons aren't required; they are added to make it easier to read the line.)

Painting a Filled Area

DRAW's use of color isn't limited to specifying the border of an object. You can also use the P (paint) option shown in Table 8-3 to fill anything inside an enclosed area with one of the available foreground colors. All you have to do is enter the P command, followed by a numeric expression representing the color code; then tell the PCjr at what color of border to stop painting. (P requires both parameters; otherwise, an error will result.) With the statement 100 DRAW "P4,15", a previously enclosed area will be painted red until a white border is encountered. Painting always begins at the LRP. Consequently, you may have to offset the LRP if it is on a line. See Figure 8-7.

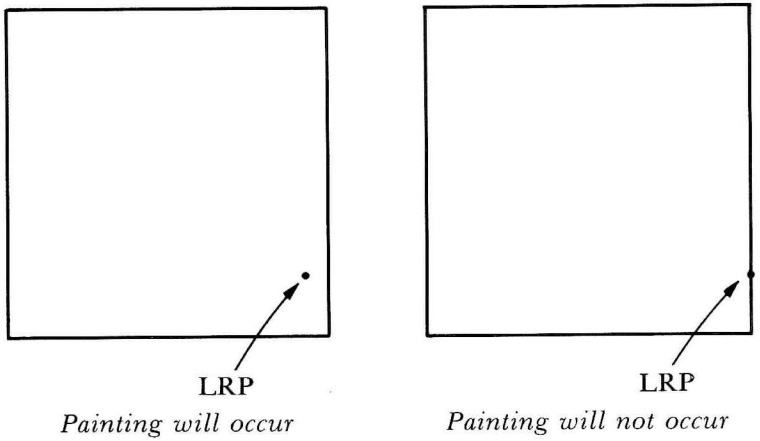


Figure 8-7. *Offsetting the LRP for the P (Paint) option*

For instance, to paint the hull of the boat red, add line 45 and modify line 60.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 BOAT$ = "R60 U40 G30 R25 D10 R30 G20
   L45 H20"
45 P$ = "BM+6,+2 P4,15"   'Offset LRP and
                           paint red
50 PSET (X,100)
60 DRAW BOAT$ + P$         'Draw boat and
                           paint it
70 X = X + 30
80 IF X > 200 THEN END
90 CLS
100 GOTO 50
110 END
```

Notice how P appears in this example. The paint must be specified in its "own" string. You cannot simply add the P option to the draw string. To draw and paint without adding another program line, you can *concatenate,* or join the two strings with +.

Program Note: In this version of the program, the boat takes longer to move across the screen because it is both painted and drawn. Paint requires offsetting the LRP—meaning that the LRP can't be on a line for painting to begin, so it must be set in an open area. Because of this, the final X,Y point for painted and unpainted boats may be slightly different.

Making Objects Larger or Smaller

In our previous example it took longer to move the boat across the screen when the boat had to be painted. Obviously, if the boat were smaller, it would take less time to paint (and conversely, if the boat were bigger, more time would be required). Furthermore, because it takes less time to paint, a smaller boat will move faster. It is easy to reduce or enlarge the size of the boat or any object created by DRAW without formulating different draw values and coordinates. Just use the S (scale) subcommand described in Table 8-2.

The PCjr "scales" the number of pixels used to create the object. For example, if you want a line twice as long as the current line, the computer will double the number of pixels you specify. On the other hand, if you want a line half as long, the PCjr will use half as many pixels.

Objects are scaled in multiples of 4. In a normal-size object, the one pixel you specify is equal to one pixel on the screen, but this 1-to-1 relationship is expressed as 4/4. This is the default value and is used unless you specify otherwise.

- Any scaling ratio less than 4/4 will make the object smaller. For instance, 2/4 (or 1/2) will create an object that is half-size. This is specified in such DRAW instructions as S2.
- Any scaling ratio greater than 4/4 will make the object larger. For example, 8/4 (or 2/1) will create an object that is double-size. This is specified in such DRAW instructions as S8.

For accuracy and convenience, use scales between 0 and 252 that are multiples of 4. We have described scaling ratio in fractional terms where the denominator (bottom number) is always 4 and the numerator (top number) is always a multiple of 4. This is important to remember.

So how do you use the S subcommand in a program line? Just enter S followed by a numeric expression that represents the numerator of the scaling ratio. See Figure 8-8.

100 DRAW "S8"	100 DRAW "S2"
Double-size	*Half-size*

Figure 8-8. *Setting scale to double- and half-size*

To see the S command at work, try the next program. When you run the program, squares will be drawn starting at the lower left corner of the screen. Squares that are smaller than normal (defined as S4) will be yellow, normal-size squares will be red, and squares that are larger than normal will be blue.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 A = 10
50 C = 14    'Small squares set to yellow
60 L = 30    'Length of lines
70 SCALE = 1
80 S$ = "S" + STR$(SCALE)
90 BOX$ = "C=C; BM160,100 U=A; R=A; D=A;
   L=A;"
100 IF SCALE = 4 THEN C = 4
             'Normal size will be red
110 DRAW S$ + BOX$
120 SCALE = SCALE + 1
130 IF SCALE > 4 THEN C = 1
             'Large squares set to blue
140 IF SCALE > 40 THEN 160
150 GOTO 80
160 END
```

If you use a variable to represent a scale factor, all of the rules concerning DRAW and variables discussed previously apply. This means an equal sign must precede the variable in the instruction and a semicolon must follow it. For instance,

```
40 D = 50
50 X = 12
60 DRAW "S=X; M160,100 U=D; L=D; D=D; R=D;"
```


Changing the Drawing Angle

In all the previous programs, drawing has been done at "normal" angles. However, two options—A and TA—let you change the displacement of the drawing angle (see Table 8-2). Normally, when you specify that a line is to be drawn "up," it is drawn perpendicular to the Y-axis. When you change the angle displacement, an "up" line may look as if it is going sideways or down. Changing the angle is useful when you want to create objects like spirals or mazes.

The A Option

The A option has four pre-set angles that are specified by a numeric expression from 0 to 3. See Figure 8-9.

To use this option, simply enter A followed by the angle code. Since the code can be a numeric expression, variables must follow DRAW's

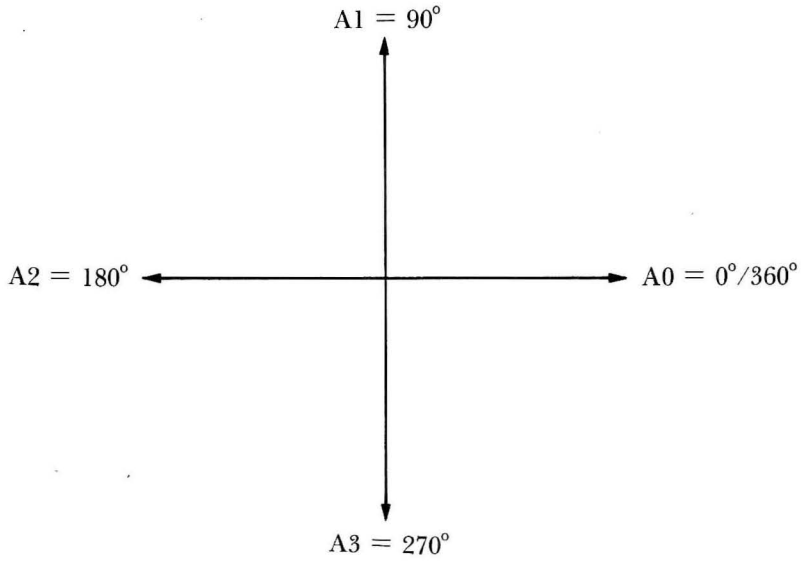


Figure 8-9. *Angles and codes for DRAW's A option*

rules for handling variables: an equal sign must precede the variable and a semicolon must follow it. See Figure 8-10.

When you turn on the PCjr, A0 (0 degrees) is used. This is the angle displacement for the following program. When you run it, an arrow will be drawn that points to the top of the screen.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 ARROW$ = "U30 NG10 F10"
50 TURN$ = "A0"
60 DRAW TURN$ + ARROW$
70 END
```

Notice that line 50 contains the A subcommand. Change **A0** to **A1** and run the program. This time the arrow will point to the left.

The following version of the program changes the angle displacement when you press an arrow (cursor movement) key. When you run the program, an arrow will be pointing up. Press an arrow key and ENTER to change the angle of the draw. Press Q to quit.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 KEY (11) ON: KEY (12) ON: KEY (13) ON
   : KEY (14) ON
40 ARROW$ = "U30 NG10 NF10 D30"
50 CLS
60 ON KEY (11) GOSUB 1100: ON KEY (12) G
   OSUB 1200: ON KEY (13) GOSUB 1400: ON
   KEY (14) GOSUB 1300 70 DRAW ARROW$
80 A$ = INKEY$: IF A$ = "" THEN 80
90 IF A$ = "Q" OR A$ = "q" THEN END
100 GOTO 60
110 END
1100 CLS: DRAW "A0": RETURN
1200 CLS: DRAW "A1": RETURN
1300 CLS: DRAW "A2": RETURN
1400 CLS: DRAW "A3": RETURN
```

The TA Option

The second angle displacement option is TA, which is somewhat more flexible than A. While A gives you only four pre-set options, TA lets

100 DRAW "A2"	100 DRAW "A7"
90 A = 2	90 A = 2
100 DRAW "A=A;"	100 DRAW "AA"
	90 A = 2
	100 DRAW "A=A"
	90 A = 3
	100 DRAW "AA;"
Right!	*Wrong!*

Figure 8-10. *Legal and illegal ways of using DRAW's A option*

you choose any angle between 0 and 360 degrees by actually specifying the angle in a numeric expression. You can preface the angle by + or − and the object will be drawn counterclockwise or clockwise.

To use this option, enter TA followed by the number of degrees to turn. Be sure to include a plus or minus sign if appropriate and, if you are using a numeric variable, to preface the variable with an equal sign and follow it with a semicolon. For instance, both of the following instructions will draw the subsequent line at 76 degrees:

```
100 DRAW "TA76"
90 A = 76
100 DRAW "TA=A;"
```

The next program will allow you to see how angles measure up to each other. When you run the program, you will be asked for the size of the angle you wish. Type a whole number between 0 and 360 and press ENTER. After that, the angle is drawn on an X-Y coordinate. To see another angle, press any key; to end the program, press BREAK.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 ARROW$ = "R70 NH10 NG10 L70"
50 INPUT "What size angle"; ANGLE
60 ANGLE$ = "TA" + STR$(ANGLE)
70 LINE (0,100)-(319,100)
```

```
80 LINE (160,0)-(160,199)
90 DRAW "M160,100 C4"
100 DRAW ANGLE$ + ARROW$
110 LOCATE 23,1
120 PRINT "Press any key to continue..."
130 A$ = INKEY$: IF A$ = "" THEN 130
140 GOTO 30
```

Assigning and Executing Substrings

Throughout this chapter, we've shown increasingly more efficient ways to write graphics programs. The use of variables and relative moves are two methods we have emphasized. However, DRAW has another very powerful option that enables you to write shorter and more flexible programs. This is the X (execute a substring) option (listed in Table 8-3).

X allows you to assign a DRAW instruction set (substring) to a string variable and then execute the substring inside another (primary) instruction string. Once execution of the substring is complete, control returns to the primary instruction string. You derive maximum flexibility from your program with a minimum of work. The only rule to follow is that a semicolon must always follow the substring.

The sailboat we created earlier in the chapter provides a good example for using X. As you recall, the boat had two parts—the hull and the sail. The following version of the boat program uses one substring to draw two sails. After each execution of SAIL$, control returns to the primary draw string.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 SAIL$ = "U40 G30 R25 D10"
50 DRAW "R31 XSAIL$; R31 XSAIL$; R31 G20
   L45 H20"
60 END
```

DRAW and the Aspect Ratio

When DRAW is used in some situations, lines seem either shorter or longer on one axis than on the other—depending on the SCREEN you are using. This is why an exact square may look like a rectangle with

unequal sides, even though you have specified equal lengths when you created it. In medium and high resolution, a line ten pixels long, for instance, will appear to be longer on the Y-axis than on the X-axis. In low resolution, just the opposite will be true. This is because pixels are not exact squares. They are rectangles with different heights and widths. The difference in height and width is referred to as the height/width ratio or aspect ratio.

For DRAW, the PCjr uses the aspect ratio of 5/3 for low, 5/6 for medium, and 5/12 for high resolution. To draw perpendicular lines that appear to be the same length on the screen, use these ratios in the form *length* × (*n*/5), where *length* is the "real" length and *n* is either 3, 6, or 12, depending on the appropriate resolution. To draw an exact square based on a 50 × 50 pixel size, use the following variation:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 A = 50
50 B = A * (6/5)
60 DRAW "R=B; U=A; L=B; D=A;"
70 END
```

For more information on aspect ratios, refer to Chapter 7.

CHAPTER NINE

Filling Areas With Color: PAINT

The last chapter described how DRAW's paint option can fill an enclosed area with an available color. In Chapter 6 you learned how to fill a box created with LINE. These methods of area filling are certainly useful, but they can't be applied to enclosed areas formed by CIRCLE or PSET or to rectangular figures created without LINE's B option. The PCjr therefore provides a general-purpose fill statement called PAINT, which can be used with any enclosed area.

THE PAINT STATEMENT

PAINT (see Syntax 9-1) is much more powerful and consequently more complex than the fill methods used by LINE/DRAW. For one thing, PAINT can add texture to a color, giving the appearance of different shades of the same color. This is called "tiling," and since the PCjr can generate over 65,000 different textures, the process can be quite complicated.

PAINT *(x,y) paint, border, background*

(*x*,*y*) is a screen coordinate that specifies the starting point for painting to begin. It should be inside an enclosed area. *x* indicates an X-axis coordinate and is a numeric expression between
0 and 159 for low resolution
0 and 319 for medium resolution
0 and 639 for high resolution.
y specifies a Y-axis coordinate. It is a numeric expression between 0 and 199 for resolutions.

paint specifies the color the area is to be filled with. It is either a numeric (integer) expression or a string expression. If *paint* is a numeric expression, it is between:
1 and 3 for SCREENs 1, 4, and 6
0 and 15 for SCREENs 3 and 5
0 and 1 for SCREEN 2.
Solid colors will be produced. If *paint* is a string expression, it is from hexadecimal 00 to hexadecimal FF and a "tiling" pattern is produced.

border specifies the color at which painting is to stop. It is a numeric expression between
1 and 3 for SCREENs 1, 4, and 6
0 and 15 for SCREENs 3 and 5
0 and 1 for SCREEN 2.
border is optional; if it is omitted, painting will stop at the first border that is the same color as *paint*.

background allows tiling over an area of the same pattern. *background* is a one-character expression that is only used if *paint* is a string expression.

Syntax 9-1. *The PAINT Statement*

Starting the Paint Process

For both solid and textured painting, the first parameter used by PAINT defines the starting point for filling to begin. This point is identified by a pair of numeric expressions representing a point on the X-axis and Y-axis coordinates. It is important to remember that the starting point must be *within* the enclosed area to be filled, as shown in Figure 9-1.

The X and Y values you specify must also be within the visible screen limits for the resolution mode being used. This means that X can be between 0 and 159 for low resolution, between 0 and 319 for medium resolution, and between 0 and 639 for high resolution. In all three resolution modes, Y can be greater than 199. If either X or Y is outside this range, an error will not occur, but neither will any painting action. In the same sense, if the start painting point happens to be directly on a border of the same color, painting will not occur.

Typically, the starting point is specified like this: 100 PAINT (200,10). Or if you've assigned values to the variables X and Y, a program might contain these lines:

```
50 X = 110
60 Y = 50
70 PAINT (X,Y)
```

The basic rules for identifying the starting point are that X,Y values must be inside parentheses, that a comma must separate the two values, and that the X value must always precede the Y value. Once painting begins, it will continue until a border of the same color or a specified border is encountered.

To use PAINT, you only need to identify the starting point. The PCjr will paint with the current foreground color, stopping at a border of the same color. The following program uses PAINT this way:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50
50 PAINT (160,100)
60 END
```

When you run this program, a white circle (the default foreground

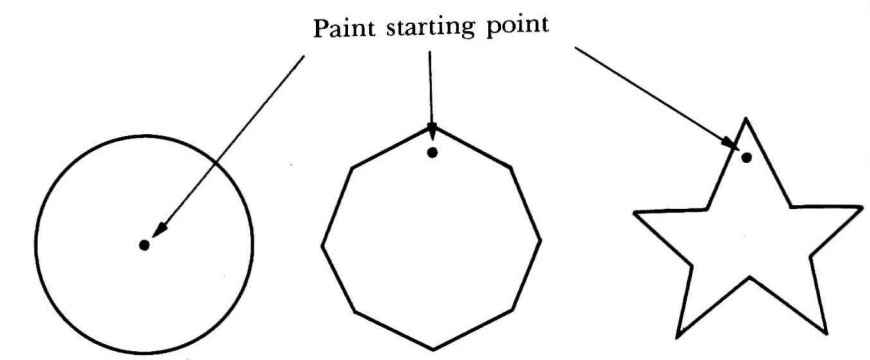


Figure 9-1. *The starting point for PAINT*

color) is drawn, and then white painting begins. Painting ends when the circle's circumference (border) is reached. What happens if the starting point for the painting operation is outside of the enclosed area? Try moving that point by changing line 50 and see.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50
50 PAINT (260,100)
60 END
```

In this case, everything except the area inside the circle is painted. The enclosed area isn't filled because it is surrounded by the color of border at which painting will automatically stop.

Be sure the enclosed area is really enclosed. If you leave any "holes" in the border, paint will leak out. The next program uses CIRCLE's start and end options to intentionally leave a small hole in the circle. As you'll see, the paint leaks out and covers the entire screen.


```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100), 50,, 1.6, 1.4
50 PAINT (160,100)
60 END
```

Leaving a hole in an object doesn't have to occur intentionally. Some objects are impossible to draw without holes, especially if they were created with statements like PSET. In Chapter 5, for instance, we created an eight-petal flower by plotting points with PSET. The flower cannot be filled because there will always be pixels that aren't set.

Painting with the default color is a simple way of using PAINT. Although you are limited to a single color at a time, you can change paint colors by changing the foreground (default) colors with the COLOR statement before executing PAINT. The following program does this. Notice that it is exactly like one of the earlier programs, but the addition of the COLOR statement in line 60 changes the default paint color to red as a new circle is drawn and painted.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50
50 PAINT (160,100)
60 COLOR 4
70 CIRCLE (40,40),30
80 PAINT (40,40)
90 END
```

Selecting a Color and Border for Solid Paints

The previous examples have left color selection up to the PCjr. But PAINT does give you the option of choosing colors. To choose a color, just include an expression that specifies an available color for the resolution you are using. If you want to use a solid color of paint, that expression must be numeric — for instance, 50 PAINT (160,100),2. Or if you are using a numeric variable to select the color, a program

might contain lines like the ones shown here:

```
40 FOR C = 1 TO 15
50 PAINT (160,100),C
60 NEXT C
```

If you try to use a color outside of the allowable range for the SCREEN you are in (color 14 in SCREEN 4, for instance), an error will occur.

Stopping Paint at a Border

Specifying a paint color isn't enough; you must also specify the color of border at which painting is to end. Otherwise, paint will overrun the border and fill the entire screen. In the next program, a blue circle is created and painted red. But because the color of border to stop at has not been specified, paint overruns the border and subsequently covers the entire screen.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50,1
50 PAINT (160,100),4
60 END
```

To stop the painting process within the boundaries of the circle, use the same color code for PAINT *border* as you did for CIRCLE *color*. That means changing line 50 to the following:

```
50 PAINT (160,100),4,1
```

This rule holds true for every graphics statement used to create an object: the PAINT *border* must match the *color* used to draw a figure if painting is to be confined to an enclosed area.

If you are using a 2- or 4-color resolution mode, PAINT should be used in conjunction with COLOR, PALETTE, or PALETTE USING to make additional colors available.

Selecting a Color and Border for Textured Paints

Another way to fill an area with color is to use a pattern or texture technique called *tiling*. Tiling is so named because it is similar to a tile on a wall or floor of a house. The tiles that cover a kitchen floor, for instance, all have the same pattern. You choose the pattern and color of tile you want and use that pattern over and over until the entire floor is tiled. The same principle applies to PAINT tiling. You choose the color and pattern of tile you want to fill an area with and repeat that pattern until the area is filled. The difference is that the PCjr allows you to create your own combinations of patterns and colors. In total, there are more than 65,000 combinations you can use. As with solid painting, the border for texture painting must be specified.

Consider the following example of how tiling is done. Suppose that a particular area is filled with solid green paint. This means that of eight pixels in a row, all eight are green.

green green green green green green green green

Now you can add texture by specifying that every third pixel be black.

green green black green green black green green

A different pattern will appear if every other pixel is black.

green black green black green black green green

Each of these combinations will produce a different pattern.

More precisely, each pixel consists of a *byte*, and each byte is made up of eight *bits*. You can manipulate those bits so that a single pixel may have a pattern of green-black and so on within it.

This manipulation of bits may seem familiar to you. Chapter 6 described how LINE can create different *line styles* using different patterns controlled by the bits that make up a pixel. The principle is the same with tiling. The difference between line styles and tiling, however, is that line styles are limited to a single, straight line, while tiling is defined as a grid that is up to 8 pixels wide and 64 pixels high. This grid is the *tile*. See Figure 9-2.

To understand how tiling works, you must be familiar with the binary (base 2) numbering system. In binary, the number 1 turns a bit

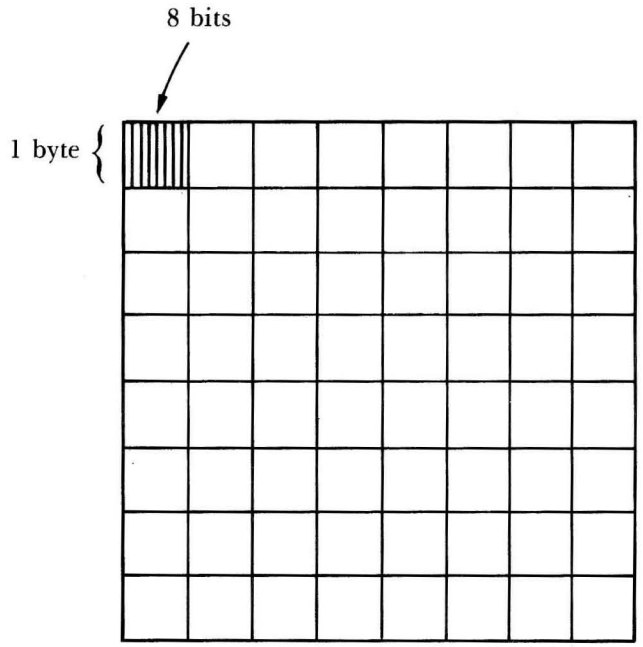


Figure 9-2. *A tile created by an 8 × 8 pixel grid*

on and the number 0 turns it off. Four bits on and four off can then be expressed this way: 1 1 1 1 0 0 0 0. While this is easy to remember, it's awkward to work with, especially if you want tile pattern 53,333 (quite a few 1's and 0's). Consequently, the PCjr lets you specify the pattern you want by using the hexadecimal (base 16) numbering system. Until you get used to it, hexadecimal may be more difficult to use, but after a while you shouldn't have any trouble with it.

Instead of the 1's and 0's used to set bits in the above example, you can use the following hexadecimal number: F0. As you can see, that is much easier to type in. For convenience, think of the binary number as pixel bits on and off (0's and 1's), and then refer to Appendix A, the Binary/Decimal/Hexadecimal conversion chart, to see which hexadecimal number to use.

Before using hexadecimal numbers, however, you must tell the PCjr that you intend to do so. Otherwise, it won't know what you are trying to do. How do you indicate to the computer that you're using a hex number? Simply enter the characters &H.

The PCjr will then treat anything that follows as a hexadecimal number as long as it is within the allowable range (that is, not greater than the decimal number 65355).

There is one other instruction required before you can create a tiling pattern. The hexadecimal number must be assigned to the CHR$ statement like this: **CHR$(&H***number*). An example is the statement 100 PAINT (160,100), CHR$(&HAB). The following program draws a circle and thenpaints it with a textured paint.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50,1
50 PAINT (160,100),CHR$(&HED),1
60 END
```

There are two elements to consider when selecting a tiling pattern—the pattern and the color. These two elements are closely related. The pattern you select will depend on the number of colors available in the different resolutions.

Selecting the Color of a Tile

Predicting the color of a tile is probably the simpler of the two elements to control if you follow the method we describe. While it is possible to compute the hexadecimal/binary values necessary to create a tile, our method is more pragmatic. It relies on the Binary/Decimal/Hexadecimal conversion chart included in Appendix A. For a discussion of how to compute the values, see *Graphics Primer for the IBM PC* (Waite and Morgan, Osborne/McGraw-Hill, 1983.)

To start with, look at the decimal number that identifies a color's absolute color code (see Table 4-1). It will be a decimal number between 1 and 15; for example, red is color code 4. Use Appendix A to find the binary equivalent of that decimal color code. See Figure 9-3.

This binary number will be listed as two groups of 0's and 1's. Take note of the second group of four digits. This is the binary color code.

Decimal	Binary	Hexadecimal
4	0000 0100	04

Figure 9-3. *Sample equivalences from the conversion chart*

Finally, use the hexadecimal equivalent of the color code in the PAINT string along with CHR$(&H).

Binary numbers that have the same last four digits will be the same color. For example, any binary number that ends in 0100 will be a variation of red. See Figure 9-4.

Selecting the Pattern of a Tile

The pattern of a tile is stored in the first four digits of the binary number. See Figure 9-5.

Once you have determined the pattern created by the first four digits (use the hunt and guess method for now), you can pick that pattern out of the Binary/Decimal/Hexadecimal conversion chart and repeat it in different situations. In Figure 9-6, for instance, two sets of binary numbers are listed. The first four numbers in one set have the same pattern (1000) but different colors. The other set has the same color (0010) but different patterns.

Decimal	Binary	Hexadecimal
4	0000 0100	04
196	1100 0100	C4
244	1111 0100	F4

Figure 9-4. *Equivalences and patterns for the color red*

1000 1011
1000 1101
1000 1110
1000 1111

Pattern Color

Figure 9-5. *Patterns versus color in a binary number sequence*

You can experiment with hex numbers in the following program. When it asks **WHAT IS THE HEX NUMBER?**, type in a number like **&H2D** or **&H56** and press ENTER. Remember that in some pattern/color combinations, the colors may not seem quite true. This may be because of your monitor, tv, or a factor of the tile; the "real" color may be masked out by the pattern. When you run the program, examine patterns that are alike but that are made up of different colors, and vice versa. Press BREAK to end the program.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 INPUT "What is the hex number";X
50 CIRCLE (160,100),50,1
60 PAINT (160,100),CHR$(X),1
70 GOTO 30
```

1000 1011	0010 0010
1000 1101	0011 0010
1000 1110	1010 0010
1000 1111	0111 0010
Same pattern, different color	*Different pattern, same color*

Figure 9-6. *Pattern/color combinations*

Let's recap the steps necessary to select a tile:

1. See Table 4-1 for the decimal color code of a color.
2. Check the Binary/Decimal/Hexadecimal conversion chart in Appendix A for the binary equivalent of the color (the last four digits).
3. Determine the binary representation of the pattern you wish to use with the color (the first four digits).
4. Determine the hexadecimal equivalent of the binary number. (See Appendix A.)
5. Include this hexadecimal number in the string CHR$(&H) as the *paint* option in the PAINT statement.

The tiling patterns and examples we have used so far have referred to a pattern in a single row across an object. That single row is then repeated to cover the entire figure. As we said, a tile consists of a grid that is 8 pixels across and as many as 64 rows of pixels down. See Figure 9-7.

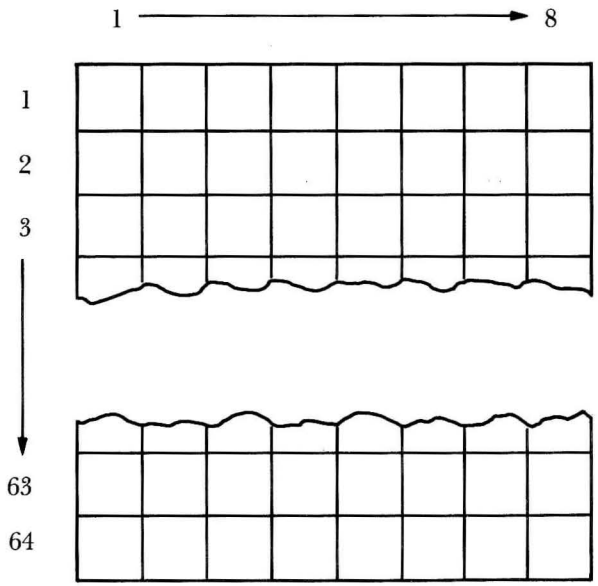


Figure 9-7. *Maximum and minimum size of a tile grid*

Creating Titles With Multiple Patterns

If you use a single CHR$(&H) string, your grid will consist of a single row—the top row—which is repeated over and over until an area is filled. This isn't very different from solid paint. However, you can begin defining the subsequent rows of the grids simply by adding CHR$(&H) instructions to the first one. A single PAINT statement may contain as many as 64 CHR$(&H) instructions.

For example, tiling data might be concatenated like this: 100 PAINT (160,100),CHR$(&H84) + CHR$(&HB1) + CHR$(&HBE). This tiling pattern is based on a grid that is eight pixels wide by three rows high. When you execute a line like this, a pattern that has a patterned row of red, a patterned row of blue, and a patterned row of yellow is displayed. At that point, the rows begin repeating until the area is filled. See Figure 9-8.

The next program creates the red, blue, and yellow pattern. However, since it is sometimes difficult to see the narrow rows of colors and

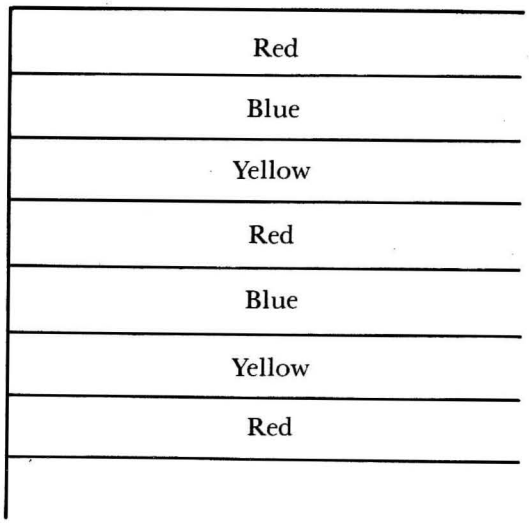


Figure 9-8. *Sample three-row tile pattern*

patterns, we will make this a tile grid with rows—three each of red, blue, and yellow.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),150,1
50 PAINT (160,100), CHR$(&H84) + CHR$
   (&H84) + CHR$(&H84) + CHR$(&HB1) +
   CHR$(&HB1) + CHR$(&HB1) + CHR$(&HBE)
   + CHR$(&HBE) + CHR$(&HBE), 1
60 END
```

Assigning String Expressions to Variables

You may have already discovered that entering a long sequence into a PAINT instruction over and over is cumbersome. The PCjr allows you to assign a series of string expressions to a variable and then execute that variable inside a PAINT instruction. For instance, you might want to modify our previous example like this:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100),50,1
50 TILE$ = CHR$(&H84) + CHR$(&H84) +
   CHR$(&H84) + CHR$(&HB1) + CHR$(&HB1)
   + CHR$(&HB1) + CHR$(&HBE) + CHR$
   (&HBE) + CHR$(&HBE)
60 PAINT (160,100), TILE$, 1
70 END
```

You can concatenate string variables as well. Add some different tiles to the display by making the following program changes:

```
55 TILE2$ = CHR$(&HCB) + CHR$(&HCB) +
   CHR$(&HF8) + CHR$(&HF8)
60 PAINT (160,100), TILE$ + TILE2$, 1
```


Tiling Over Another Tile

In most cases, you can tile over an area that has previously been tiled. This is useful and sometimes necessary in games and graphs when situations or data change, and you want to accentuate the differences.

However, there is a situation that may occur where special considerations must be taken. This is that tiling will not continue if one hexadecimal character of the new tile matches one character of the old tile, but does not match two consecutive characters in the tiling string. (Or to put it another way, *if* at least one hexadecimal character of the new tile matches any hexadecimal character of the old tile, *then* the hexadecimal character of the new tile cannot disagree with two consecutive characters of the old tile.)

Tiling over a tile can take place in the first set of program lines shown in Figure 9-9 because there are no matching hexadecimal characters in the two strings. In the second set of program lines, however, the hex character &H7C occurs in both lines. When the lines are executed, tiling in line 90 cannot be completed, so the program goes on to the next instruction in the program.

Incomplete tiling can be a serious problem if you want to use the same pattern but in a different color. PAINT has a way around the problem, however. The *background* option (see Syntax 9-1) allows continual new tiling over an area that has a hexadecimal character (pattern) used in the old tile. This option is a one-character hexadecimal

---

```
80 PAINT (160,100), CHR$(&H7C), 2
90 PAINT (160,100), CHR$(&HA2) + CHR$(&H34) + CHR$(&H34), 2
```

Legal!

```
80 PAINT (160,100), CHR$(&H7C), 2
90 PAINT (160,100), CHR$(&H7C) + CHR$(&H34)+ CHR$(&H34), 2
```

Illegal!

Figure 9-9. *The right and wrong way to tile over a tile*

string that effectively tells the PAINT command that it is all right to tile over the area. Run the following program to see how it is used:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 CIRCLE (160,100), 50, 2
50 PAINT (160,100), CHR$(&H7C), 2
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 PAINT (160,100), CHR$(&H7C) +
   CHR$(&H34) + CHR$(&H34), 2
80 END
```

When you execute this program, the tiling pattern in line 70 is not completed because &H7C occurs in both lines and because &H34 occurs consecutively in line 70. If &H7C didn't match, or if &H34 were not used twice in a row, then the new tile could be completed. The new tile can also be completed if the background option is added to line 70. Change line 70 to the following:

```
70 PAINT (160,100),CHR$(&H7C) + CHR$(&H34)
   + CHR$(&H34),2,CHR$(&H7C)
```

Now when you run the program, the second tiling pattern fills the entire circle.

The first tiling pattern in the previous example used a one-character string. However, the same principle holds true for patterns that are more than one string. Run the next program and compare it with the previous program.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 X=160: Y=100: C=2
50 CIRCLE (X,Y), 50, C
60 TILE1$ = CHR$(&H7C) + CHR$(&H5A) +
   CHR$(&H42)
70 TILE2$ = CHR$(&H7C) + CHR$(&H34) +
   CHR$(&H34)
80 PAINT (X,Y), TILE1$, C
90 FOR DELAY = 1 TO 600: NEXT DELAY
100 PAINT (X,Y), TILE2$, C, CHR$(&H7C)
110 END
```

CHAPTER TEN

Simulating Movement: GET and PUT

Once you've created a figure, one of the more exciting things you can do is move it around the screen. *Animation* simulates movement by re-creating the same inanimate objects at different locations. (Cartoons and video games are based upon this principle.) The most efficient method of animating objects on the PCjr uses GET and PUT. These two statements must be used together and are therefore discussed together throughout this chapter.

GET and PUT movement works as follows. First you create an object using one of the graphics commands we've already described (LINE, CIRCLE, DRAW, and so on). Then you "get" the object from one place on the screen and "put" it somewhere else.

Sounds simple, doesn't it? In principle it is, but in order to use GET and PUT effectively, you must be familiar with several programming concepts, particularly *arrays*. If you're not familiar with arrays, or have forgotten about them, read Part Six in the edition of *Hands-On BASIC* that came with your PCjr. Four chapters of that book describe arrays and how to use them.

Even though GET and PUT work closely together, they present different sorts of problems. The most difficult aspect of using GET, for instance, is determining the size of array to set up before you begin moving objects. For PUT, the effect of optional logical operators is a consideration. Be sure to pay attention to the discussions of both concepts as you read this chapter.

There is one other important note. Your PCjr recognizes and uses two different versions of GET and PUT. One type is used for moving graphics objects; those are the ones we will discuss in this chapter. The other forms of GET and PUT are used to move data into files. We won't discuss those statements here. For a description of them, see your *PCjr BASIC Reference Manual.*

THE GET STATEMENT

The syntax of both GET and PUT (described in Syntax 10-1 and 10-2 respectively) resembles LINE's syntax when it uses the B (box) option. GET requires you to identify the upper left and lower right corners of a rectangle that "contains" the object to be moved. With PUT, on the other hand, you only need to identify the upper left corner of the rectangle.

GET (*x1,y1*)–(*x2,y2*),*arrayname*

x1 and *y1* are numeric expressions that specify the upper left corner of an imaginary rectangle on the display. *x1* is between
0 and 159 for low resolution
0 and 319 for medium resolution
0 and 639 for high resolution.
For all resolutions, *y1* is between 0 and 199.

x2 and *y2* are numeric expressions that specify the lower right corner of an imaginary rectangle on the display. *x2* is between
0 and 159 for low resolution
0 and 319 for medium resolution
0 and 639 for high resolution.
For all resolutions, *y2* is between 0 and 199.

arrayname is the valid name of a predefined array that stores the object bounded by the imaginary rectangle.

Syntax 10-1. *The GET Statement*

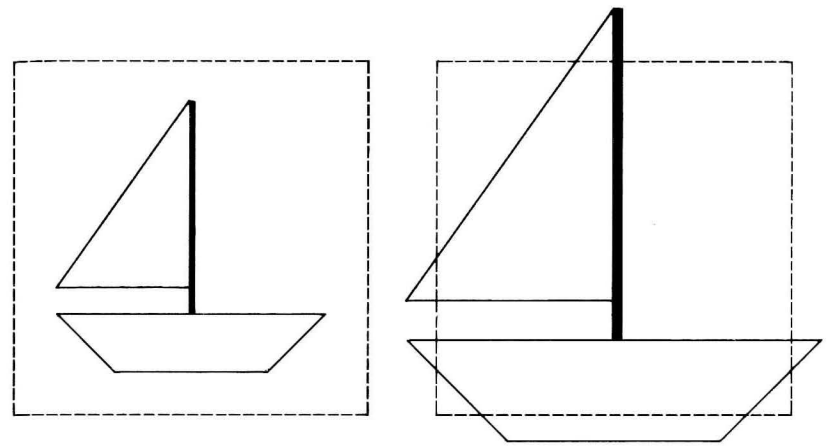


Figure 10-1. *GET's imaginary rectangle*

Even though we have referred to the diagonal points of a "rectangle," you don't actually have to draw the rectangle itself. The object you are drawing must be able to fit inside an imaginary rectangle whose size is determined by (*x1,y1*) and (*x2,y2*). Figure 10-1 illustrates this concept.

The name you assign to the array (*arrayname*) can be any valid array variable (name) described in *Hands-On BASIC.* For example, A, W$, CAR, and BOAT$ are all valid array variables.

Remember that each array variable must have a *subscript* (a number following the variable, which is enclosed in parentheses) and that if the subscript is greater than 10, you must also use the DIM statement (to *dimension* the array) elsewhere in the program.

The following program lines are examples of correct uses of the GET statement.

```
100 GET (160,100) – (180,120),A
200 GET (200,50) – (250,60),W$
100 GET (10,10) – (40,40),BOAT$
```

THE PUT STATEMENT

After an object has been identified by GET, use PUT to move it elsewhere on the screen.

Once you have defined the diagonals of an imaginary rectangle with GET (we'll call this the "source") and are ready to move the graphics contents of the rectangle somewhere else on the screen (the "destination"), use PUT to identify the upper left corner of the destination. After that, enter the name you assigned to the array containing the graphics object you have created. (This is necessary because your program may have more than one array.) Finally, include the action option if necessary. In general, *action* coordinates the movement of the object with the currently displayed colors or objects. The action operator you use depends on the effect you wish to create. For example, you can superimpose one object on top of another, reverse colors, or combine screen images.

PUT (*x3,y3*), *arrayname,action*

x3 and *y3* are numeric expressions that specify the upper left corner of an imaginary rectangle on the display. *x3* is between
0 and 159 for low resolution
0 and 319 for medium resolution
0 and 639 for high resolution.
For all resolutions, *y3* is between 0 and 199.

arrayname is the name of a predefined array that stores the object bounded by the imaginary rectangle.

action determines how the object is to be displayed on the screen. It is one of the following: **PSET**, **PRESET**, **XOR**, **OR**, or **NOT**. *action* is optional; if it is omitted, **XOR** is used.

Syntax 10-2. *The PUT Statement*

AN OVERVIEW OF GET AND PUT

As we said earlier, the data that creates an object is stored in an array. The PCjr "gets" the contents of the array (or rectangle) and "puts" it elsewhere. Figure 10-2 illustrates this activity.

To make animation appear realistic, you must clear (erase) the part of the screen the object inhabited as it moved from place to place, or else duplicate objects will be displayed. The following program illustrates these points:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DIM TEST(124)
50 CIRCLE (35,35), 10
60 GET (20,20) - (50,50),TEST
70 FOR DELAY = 1 TO 600: NEXT DELAY
80 PUT (160,100), TEST
90 END
```

When you run this program, a circle is drawn in the upper left of the screen. After a short delay, a duplicate circle is drawn near the center of the display. Movement isn't simulated because the screen contains both circles. Adding line 75 clears the screen so that it appears the circle moves from one location to the other.

```
75 CLS
```

Using Arrays With GET and PUT

There are different types of arrays you can use with your PCjr. The two most common types are one-dimensional and two-dimensional arrays. (One-dimensional arrays use a single number inside the parentheses, while two-dimensional arrays use a pair of numbers inside the parentheses.) The previous program, like most of the GET and PUT programs in this book, uses one-dimensional arrays because they are more efficient and accurate than two-dimensional arrays. Both types of arrays usually require dimensioning if more than ten elements are used.

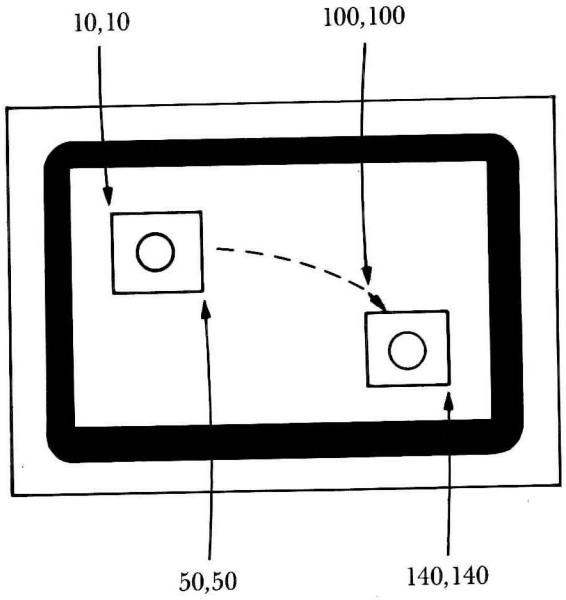


Figure 10-2. *Basic principles of GET and PUT*

One difficult problem you will encounter when using any type of array is determining the correct size to dimension an array with the DIM statement. If an array is dimensioned too small, the PCjr will display an Illegal Function Call error message. This message usually says that the error is in the program line containing the GET statement. To correct the error, however, you must modify the program line containing the DIM statement. To illustrate this, change line 40 in the previous program to **40 DIM TEST(50)** and run it. The first circle is drawn, but before it can be moved, the error message ILLEGAL FUNCTION CALL IN 60 is displayed. To eliminate the error situation, change line 40 (not line 60) back to **128**.

The following three factors influence the size of the array:

- The size of the object you are moving
- The type of variable assigned to the array
- The SCREEN mode you are using.

The influence of the first factor (the size of the object) should be apparent—the larger the physical size of the object, the larger the array must be. This means that if you increase the size of an object in one program line, you must also increase the array size in another line. Other adjustments will have to be made as well.

The second factor that affects the dimension size is the type of variable used to represent the array. The PCjr recognizes three types of numeric variables—integer, single-precision, and double-precision. The most commonly used type is the single-precision numeric variable. Consequently, it is the default value.

The criteria for choosing a variable type include accuracy, speed, and available memory. The last of these, available memory, is the most important for graphics programming. (The effect of variables on accuracy and speed is more important in high-level number-crunching programs.) Table 10-1 describes the exact memory requirements for each variable type. For most graphics applications, you should simply use the default value. You'll need to refer to Table 10-1 when determining array dimensions described in the next section of this chapter. (If you do need to use another variable type, the **DEF *variable type*** statement allows you to do so. For a description of this statement, see your *PCjr BASIC Reference Manual.*)

The third influence on array size is the SCREEN mode you use. As a general rule, a 16-color SCREEN mode requires an array two times as large as a 4-color mode, and a 4-color mode requires an array twice the size of a 2-color mode. For instance, if you dimension an array for SCREEN 2, then change to SCREEN 4, you will need to double the dimension value. In the same sense, if you go from a SCREEN mode that uses a 16K page size (SCREENs 1 through 4) to a SCREEN that uses the 32K page (SCREEN 5 or 6), you must also double the dimension value. The next series of programs provides a practical explanation of this concept.

Table 10-1. *Memory Required by Each Variable Type*

Variable Type	Number of Bytes
Integer	2
Single-precision	4
Double-precision	8

When you run the following program, the circle moves just as it should:

```
10 SCREEN 2        '2-color, 16K page size
20 CLS
30 DIM TEST (32) 'Note size of array
40 CIRCLE (35,35), 10
50 GET (20,20) - (50,50), TEST
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 CLS
80 PUT (100,100), TEST
90 END
```

The next program is exactly like the last one, except that it uses SCREEN 4 instead of SCREEN 2.

```
10 SCREEN 4        '4-color, 16K page size
20 CLS
30 DIM TEST (64) 'Note size of array
40 CIRCLE (35,35), 10
50 GET (20,20) - (50,50), TEST
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 CLS
80 PUT (100,100), TEST
90 END
```

The program again works as it should. However, if the SCREEN mode (line 10) had been changed without changing the array size (line 30), the Illegal Function Call error would have appeared.

The next version of the program is written with SCREEN 3. While it uses the same page size (16K) as SCREEN 4, more colors are available, requiring the array dimension to be doubled.

```
10 SCREEN 3        '16-color, 16K page size
20 CLS
30 DIM TEST (128) 'Note size of array
40 CIRCLE (35,35), 10
50 GET (20,20) - (50,50), TEST
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 CLS
80 PUT (100,100), TEST
90 END
```


In the following program, the CLEAR statement is added in line 5 because line 10 takes the program into SCREEN 6. Even though the page size increases (from 16K to 32K), the dimension value remains the same because the number of available colors is reduced from the previous program (from 16 to 4). However, SCREEN 5 (16-color, 32K page size) is used without increasing the size of the array.

```
5 CLEAR ,,,32768
10 SCREEN 6        '4-color, 32K page size
20 CLS
30 DIM TEST (128)'Note size of array
40 CIRCLE (35,35), 10
50 GET (20,20) - (50,50), TEST
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 CLS
80 PUT (100,100), TEST
90 END
```

You can see that there are many factors influencing the size of the arrays you use with GET and PUT. Of course, the size of the object is important, but as we have pointed out, the type of variables and SCREEN also need to be considered. Certainly, you should not expect to change the SCREEN without also changing the size of the array.

How Big Should an Array Be?

Since one-dimensional arrays are the most efficient type of arrays to use for graphics programming, use the following formula to determine the DIM requirements:

$$value = \frac{(4 + \text{INT}\ ((x \times bits + 7) / 8) \times y)}{variable\ type}$$

In this formula, x is the number of X-axis pixels on the imaginary rectangle and y is the number of Y-axis pixels. *bits* is the number of bits per pixel for the SCREEN you are using and is either 1, 2, or 4 (see Table 10-2). *variable type* is the number of bytes required by the variable and is either 2, 4, or 8 (see Table 10-1). The result of this formula is equal to *value*, and *value* is then used as the subscript in a

Table 10-2. *Number of Bits per Pixel in Each SCREEN*

SCREEN	bits per pixel
2	1
1, 4, 6	2
3, 5	4

one-dimensional array, as shown:

DIM *arrayname (value)*

Let's use this formula to show how the program line **30 DIM TEST (64)** was derived in the following example:

```
10 SCREEN 2        '2-color, 16K page size
20 CLS
25 DEFINT T        'Integer variable
30 DIM TEST (64)   'Note size of array
40 CIRCLE (35,35), 10
50 GET (20,20) - (50,50), TEST
60 FOR DELAY = 1 TO 600: NEXT DELAY
70 CLS
80 PUT (100,100), TEST
90 END
```

(**Programming Hint:** You'll find that it is usually a good idea to make the array subscript a multiple of 8. For instance, if the formula results in a *value* of 30, round this value off to 32. Or if the result is 117, round it off to 120 and so on.)

1. The value of *x* and *y* is calculated by subtracting 50 − 20 = 30 (*x2* − *x1* = X) and 50 − 20 = 30 (*y2* − *y1* = Y). These are then substituted into the following formula:

$$value = \frac{(4 + \text{INT}\,((30 \times bits + 7) / 8) \times 30)}{variable\ type}$$

2. Next, the value of *bits* per pixel is included. Table 10-2 shows that SCREEN 2 uses 1 bit per pixel. When calculated, this results in:

$$value = \frac{124}{}$$

3. Line 25 defines the variable T (the first letter of the name TEST) as an integer and as Table 10-1 shows, integers require **2** bytes per element. Therefore, **2** is divided into **124**.

$$value = \frac{124}{2}$$

This results in

value = 62

4. Finally, round *value* off to the nearest multiple of 8 and use it in the DIM statement.

30 DIM TEST (64)

When the formula is used this way, it successfully accounts for all the factors that influence the size of the array—the X,Y size of the object, the SCREEN you are using, and the type of variable representing the array.

In Chapter 8, we used DRAW to create a small sailboat and simulated motion by drawing it over and over across the screen. As we said then, the use of GET and PUT is a much more sophisticated animation technique that results in more satisfactory motion. The next program uses the same DRAW string to create a boat, but uses GET/PUT to move it across the screen.

Notice that this movement is less jerky yet faster than previous ways of moving the boat. Line 130 is followed by an exact duplicate instruction in line 140, thereby reducing the flicker that can occur with GET/PUT movement. (For an explanation of this principle, see the discussion of "XOR" later in this chapter.) When program execution begins, the boat is drawn before it is "launched." Also, the boat is drawn at 1/4 scale. If you change the scale, you'll also have to change the dimension of the array, and so on. By changing the STEP value in line 120, you can increase or decrease the boat's speed.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 BOAT$= "BM10,10 C4 S1 R60 U40 G30 R25
   D10 R30 G20 L45 H20"
50 DIM BOAT$ (120)
60 DRAW BOAT$
70 X = 10: Y = 5
```

```
90 CLS
100 LINE (0,20) - (319,20)
110 PAINT (160,100),1,15
120 FOR COURSE = 0 TO 280 STEP 5
130 PUT (X + COURSE, Y), BOAT$
140 PUT (X + COURSE, Y), BOAT$
150 NEXT COURSE
160 END
```

Array Size in SCREEN 6

When moving images in SCREEN 6, you must make special allowances for irregularities ("bugs") in PCjr BASIC (version 1.1). In SCREEN 6, GET gets an area twice as wide as requested. For example, the statement

GET (0,0)−(9,9),ARRAY

should store a 10 × 10 array. However, it in fact stores a 20 × 10 array. To compensate for this bug, you will have to use a different formula than the one presented in the previous section to calculate the number of bytes required to store a GET array. The formula for SCREEN 6 is

value = 4 + INT ((7+*x-pixels*)/8)×2×*y-pixels*

Our programs in Chapter 14 through 20 take this problem into consideration. Future versions of PCjr BASICA may fix this bug, requiring program modification to the routines in the latter part of this book.

PUT'S ACTION PARAMETERS

The action parameter you decide to use with PUT will depend upon the type of effect you wish to create. There are five options: PSET, PRESET, XOR, OR, or AND; XOR is the default operator. Depending on the option you choose, you can reverse the color of images, combine them, or superimpose one image on top of another.

One of the most important considerations in choosing an operator is the effect of the background color. There are three possibilities: you don't want the background color changed by an object moving across it; you do want the background changed by an object moving across it; or you don't care whether the background is changed or not. There

and you may or may not want it changed when it encounters a new background color. PUT's action parameters let you control these elements.

- XOR leaves the background color intact but changes the foreground color if necessary.
- PSET leaves the foreground color intact but changes the background color.
- PRESET changes both the foreground and the background color to create a negative image of the figure.
- OR leaves the background color intact but may change the foreground color in some situations.
- AND changes both the foreground and the background color.

The following program draws a circle, then gets the circle and puts it in an area filled with red. Run the program five times, each time using a different PUT action parameter in line 130, and watch the differences. Once you are familiar with the pattern, change the circle itself to red (add ,4 to line 70). Notice the different results. Later change the circle to blue and watch what happens then.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 DIM TEST (128)
50 VIEW (10,10) - (60,60),,15
60 WINDOW SCREEN (10,10) - (60,60)
70 CIRCLE (35,35), 10  'Change color
   of circle here
80 GET (20,20) - (50,50), TEST
90 FOR DELAY = 1 TO 600: NEXT DELAY
100 CLS
110 VIEW (100,100) - (160,160),4,15
120 WINDOW SCREEN (100,100) - (160,160)
130 PUT (115,115), TEST, XOR 'Substitute
    options here
140 VIEW
150 END
```

XOR

Of all the action parameters you can use with PUT, XOR (short for "exclusive OR") is probably the one you'll use the most frequently

(that's one reason it is the default parameter) because it has two highly useful features. The first time XOR is used at a screen position, the pixels are set (turned on), while the second time it is used in the same location, the pixels are reset (turned off). This makes it easy to erase the portion of the screen that contains an object as that object moves. It is why the boat program has duplicate lines 130 and 140. Line 130 draws (sets) the boat, and line 140 erases (resets) the boat before "putting" the next one. Eliminate line 140, run the program, and notice how much more jerky the movement is. Use this duplicate line technique as much as possible for smooth movement.

The second important feature is that XOR also lets you move an object to different screen locations without disturbing the background. If you are moving an object across a screen that changes from red to blue, for instance, the background won't change, but the object will. See Figure 10-3. You saw this when you drew a red circle on the red background in the previous program; the circle was still visible because it changed from red to black.

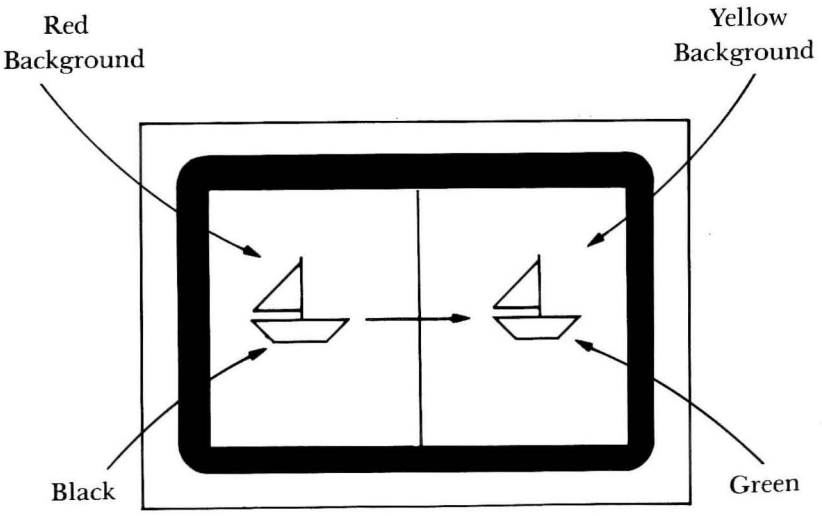


Figure 10-3. *XOR changes foreground without disturbing background*

PSET and PRESET

There are advantages and disadvantages to using PSET instead of XOR. On the positive side, PSET is somewhat faster than XOR because an additional PUT is not required. On the negative side, PSET will disturb the background color by leaving an impression on the screen once you've moved it. The deciding factor is whether or not altering the background matters to you. Since PUT with PSET simply stores the object on the screen, it is the "true" opposite of GET.

If you only want to draw the object on the screen and don't care about what is already there, PSET is the option to use. It will draw on top of, and consequently replace, any screen display. In some instances, this is desirable; in others, you may not want to erase or replace what is on the screen, but you do want to draw "on top" of it.

In the sample program, for instance, the entire array storing the circle simply blocks out the red paint of the destination viewport. Since PSET can't compensate for putting a visible red circle on a red background as XOR does, it uses the brute force method by writing over the red fill.

The concept of "positive" and "negative" films can help you understand PSET and PRESET. A positive film shows an image the way it appears to us—the moon is white and the night sky is black. On a negative image, however, the colors are reversed—the moon is black and the night sky is white. In conceptualizing images that appear on the PCjr screen, think of PSET as creating positive images and PRESET as creating negative images. In the example program above, the PSET action will put a white circle on the black background, while PRESET will reverse the image by putting a black circle on a white background. See Figure 10-4.

OR and AND

OR will superimpose an object on any color or object without changing the background. It is different from XOR, because if the background is the same color as the object being put on it, OR will not change the foreground color to compensate. That is why, in the sample program above, you couldn't see the red circle on the red background.

If the object being put with OR is the same color as the foreground, then the foreground color is put on top of the background without affecting either. The catch, however, is when the object being put is

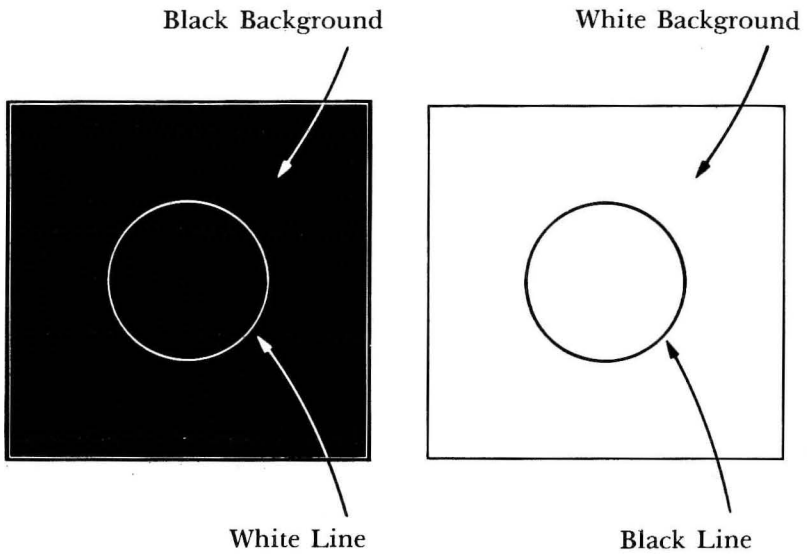


Figure 10-4. *Positive effect of PSET and negative effect of PRESET*

displayed in a different color from either the foreground or background. If this occurs, the object changes to a third color (that is, a color different from either the background or the foreground) at the destination location.

To illustrate this, run the previous sample program and put a white circle (the foreground color) on the red fill; it shows up perfectly. Next change line 70 so you put a red circle on the red background; you can't see the circle even though it was put there. Finally, change line 70 to put a blue circle on the red background (leaving white as the current foreground); in this case, the destination circle is magenta.

AND is probably the least used of all the operators. You should only use the operator AND when you want to put an image on top of another image. It will not erase the source image, nor will it clear the destination. It simply takes the stored data and duplicates it.

C H A P T E R E L E V E N

Moving Graphics Between Pages: PCOPY

While GET and PUT are probably the best way to simulate motion with your PCjr, PCOPY provides another way to experiment with animation. This statement is especially useful in animation programs that require maintaining backgrounds but changing foregrounds. Instead of redrawing the background for each scene, you can simply PCOPY (short for "page copy") the background to another page, then draw the new foreground scene. This speeds up program execution significantly.

PCOPY is also useful for programs that don't require animation. Page copying offers an easy way to compare graphics data. Instead of drawing several elements of a graph on a single screen, you can create them on separate pages, then copy one to another for comparison. This is somewhat like placing transparent overlays on top of one another.

THE PCOPY STATEMENT

PCOPY (described in Syntax 11-1) allows you to quickly copy the contents of one memory page to another. To use PCOPY effectively, you must keep track of memory requirements, the number of pages available, and the page size for each SCREEN mode. Table 11-1 lists the

Table 11-1. *SCREEN Modes and Page Comparisons*

SCREEN Mode	Maximum # of Pages	Page Size
0	48/24	2K/4K
1, 2, 3, 4	6	16K
5, 6	3	32K

numbers and size (in bytes) of pages available in the different SCREEN modes.

The pages in each of the graphics display modes 1 through 4 require 16K of memory per page; the pages in SCREENs 5 and 6 require 32K each. Therefore, you must be sure to CLEAR enough memory at the beginning of your program to accommodate the number of pages you will be using. To determine how much memory to set aside, multiply the number of pages by the page size. For instance, if you want to use two pages in SCREEN 3, multiply 16K × 2 for 32K (then multiply 32 × 1024 for the exact number to include with your CLEAR statement). You will also need to be familiar with both *active* and *visual* pages, which are used with the SCREEN statement. See Chapter 3 for a description of both SCREEN and CLEAR.

Remember that page numbering starts at 0, not 1. The page numbers

PCOPY *source page, destination page*

source page is the memory page that contains the graphics object to be copied. It is a numeric expression between 0 and 6 depending on the graphics SCREEN being used.

destination page is the memory page that is to receive the graphics object being copied. It is a numeric expression between 0 and 6 depending on the graphics SCREEN being used.

Syntax 11-1. *The PCOPY Statement*

available for a SCREEN that has six pages is from 0 to 5. If you try to display page 6, an error will occur.

The following program illustrates how PCOPY can offer a programming shortcut in some situations. In this example, we use a simple painted circle as the background scene to be copied from one page to another. Notice that the circle is only drawn and painted once; it is then copied onto different pages. This speeds up the drawing time for all but the first page. The foreground, a message identifying the current memory page, is what changes in this program.

As you can imagine, this can be a real time-saver for an intricate and complicated pattern. In most cases, you will find it advantageous to draw and paint on an active but invisible page, then use PCOPY to copy it to a visible page at a later time. In this way objects can become instantly available.

```
10 CLEAR ,,,98304  'Sets aside
   memory for up to 6 pages
20 CLS
30 FOR PAGE = 0 TO 4
40 SCREEN 1,,PAGE,PAGE
50 CIRCLE (160,100),60
60 PAINT (160,100),2,3
70 LOCATE 13,17: PRINT "Page " PAGE
80 PCOPY PAGE, PAGE + 1 'Copies from
   one page to the next
90 FOR DELAY = 1 TO 600: NEXT DELAY
100 NEXT PAGE  'Copy to and display
    next page
110 END
```

PAGE FLIPPING

When we introduced memory pages in Chapter 3, we mentioned that they can be used to simulate animation through page flipping. This technique is somewhat like drawing an object on several pages of a tablet, but at slightly different positions. You then flip the pages so that the object appears to be moving. Page flipping is, of course, the basis for most cartoon animation.

The next program uses this technique to make a yo-yo appear to go up and down. While the program doesn't use PCOPY, it does make

ample use of memory pages and illustrates how page flipping is one way to simulate motion.

```
10 CLEAR ,,,98304 'Set aside memory
   for 6 16K pages
20 FOR PAGE = 0 TO 5 'Use all 6
   available pages
30 SCREEN 2,,PAGE,0 'Select high
   resolution
40 CLS
50 LINE (320,0) - (335,20 + (PAGE-1) * 2
   5)  'Draw cord
60 CIRCLE (320,PAGE*25),25 'Draw yo-yo
70 PAINT (320,PAGE * 25),1,15 'Paint
   yo-yo
80 NEXT PAGE
90 FOR PAGE = 1 TO 5: GOSUB 120: NEXT
   PAGE  'Goes down
100 FOR PAGE = 4 TO 1 STEP-1: GOSUB 120
    : NEXT PAGE  'Goes up
110 GOTO 90
120  SCREEN 2,,PAGE,PAGE
130 FOR DELAY = 1 TO 80: NEXT DELAY
140 RETURN
150 END
```

CHAPTER TWELVE

Redefining the Screen: VIEW and WINDOW

In the previous chapters we have assumed a maximum size of the screen's drawing area and maximum values for the X,Y coordinates of the numbering system.

All of our example programs have used a common coordinate system. In this system, the smallest X,Y values (0,0) have been in the upper left of the screen, while the greatest X,Y values (for high resolution, that is, 639,199) have been in the lower right.

However, you can redefine both the size of the drawing area on the display and the numbering system used for screen coordinates. In fact, several "miniscreens," each with its own coordinate system, can be displayed.

Why would you want to redefine the screen size? For one thing, miniscreens (generally referred to as *windows*) allow you to display several different graphics objects on the screen at once. Each of these miniscreens is independent of the others, so you can, for instance, color the background of one window without affecting the background colors of the other windows. This is the principle behind many sophisticated programs that use the windowing approach. Figure 12-1 illustrates this concept.

Redefining the screen coordinates can make pixel (X,Y) identification easier. When you do redefine the screen, the coordinate system can be based on the object you're drawing. This system is generally referred to as the *object* or *image coordinate system*. With it, you may have

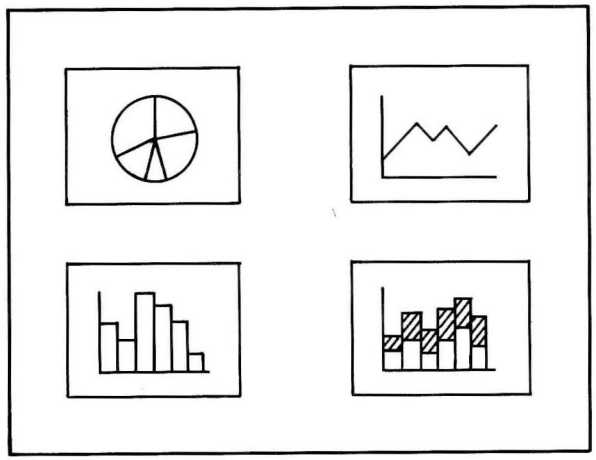


Figure 12-1. *Windows and viewports*

several windows, each with a location identified as 0,0, as shown in Figure 12-2.

Since the computer must keep track of the "real" coordinates for the miniscreens you define, a process called *viewing transformation* is automatically performed by the PCjr. This process might take one of the 0,0 points, for example, and adjust it to the absolute screen coordinate of 150,40. Since viewing transformation is totally transparent to you, you can define the screen coordinates without worry, and the computer will make any adjustments needed to accommodate the new coordinates.

The two PCjr statements used to control viewing areas are VIEW and WINDOW.

THE VIEW STATEMENT

VIEW (shown in Syntax 12-1) creates a *viewport* or active area that is associated with the screen coordinate system (that is, a miniscreen). When a viewport is defined, the viewport is the only area on the screen

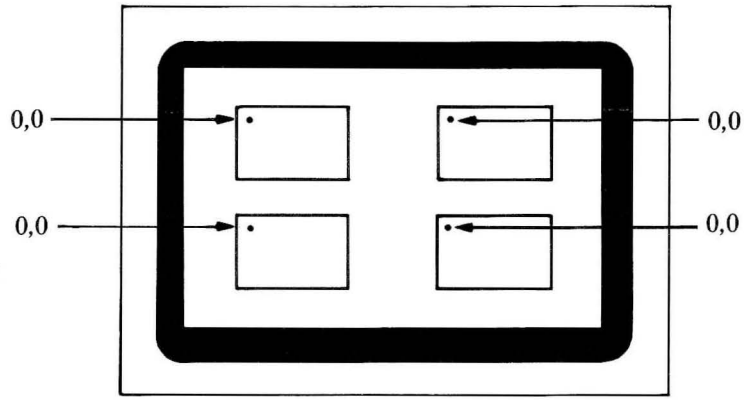


Figure 12-2. *X=0, Y=0 in the object coordinate system*

where drawing can take place. When you start up the PCjr, the viewport is, in effect, the entire screen.

Setting Up the Viewport

To designate an area of the screen as a viewport, you can use VIEW in much the same way the LINE statement's B (box) option is used. This means that you define a rectangular area on the screen by specifying its diagonal corners. After that, all your graphics commands will affect only the rectangular area identified by the diagonal points—the viewport. The following program, for instance, sets up a viewport in the lower right portion of the screen:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 VIEW (160,100) - (310,190)
```

When you run the program, nothing seems to happen—the screen is

VIEW (*x1,y1*)–(*x2,y2*), *background, border*
VIEW SCREEN (*x1,y1*)–(*x2,y2*), *background, border*

x1 and *y1* specify the upper left corner of a rectangle that represents the new viewport. *x1* is a numeric expression between:
0 and 159 (low resolution),
0 and 319 (medium resolution), and
0 and 639 (high resolution).
y1 is between 0 and 199.

x2 and *y2* specify the lower right corner of a rectangle that represents the new viewport. *x2* is a numeric expression between:
0 and 159 (low resolution),
0 and 319 (medium resolution), and
0 and 639 (high resolution).
y2 is between 0 and 199.

background specifies the background color of the new viewport and is a numeric expression between:
0 and 3 for SCREENs 1, 4, and 6,
0 and 15 for SCREENs 3 and 5,
0 or 1 for SCREEN 2.
background is optional; if omitted, the current background color is used.

border specifies the color of the border (frame) that surrounds the viewport. It is a numeric expression between:
0 and 3 for SCREENs 1, 4, and 6,
0 and 15 for SCREENs 3 and 5,
0 or 1 for SCREEN 2.
border is optional; if omitted, a boundary is not drawn.

SCREEN specifies whether the coordinates in the viewport are relative to the viewport or whether they are absolute screen coordinates. SCREEN is optional; if omitted, the X,Y screen values are relative to the viewport. If SCREEN is used, the viewport's coordinate system will be the standard values for the current resolution mode.

Syntax 12-1. *The VIEW Statement*

black and the OK prompt is in the upper left of the display. Type something on the keyboard (your name, for instance) and press ENTER. Then type **CLS** and press ENTER again. Is the screen erased as you thought it would be? Now try to draw a circle in the upper right corner by typing **CIRCLE (300,50),10** and pressing ENTER. Again, nothing seems to happen.

That's because all graphics activity has been confined to a viewport area that, in this case, is bounded by an imaginary rectangle in the lower right quarter of the screen. Any X,Y coordinates specified in an instruction must fall within the viewport range, or the resulting graphics will not appear. To see what we mean, type in the statement **CIRCLE (300,150),10** and press ENTER. This time a circle appears in the lower right corner. Now when you type **CLS** and press ENTER, the circle disappears, because the active screen area (the viewport) is cleared.

Remember that you can use any valid numeric expression, including variables, when identifying the viewport diagonal points. To cancel the viewport, type **VIEW** and press ENTER. Now if you clear the screen with CLS, the entire screen will be erased.

Selecting a Viewport's Background Color

Normally, when you change the background color of the active screen, the entire screen is affected. When you change the color of the viewport's background, only the viewport itself is altered. Modify line 40 in the above program by adding a color code to the background color option. Try using blue.

```
40 VIEW (160,100) - (310,190),1
```

If nothing else, changing the background color shows you exactly what area has been defined as the viewport. Any allowed color for the current SCREEN can be used as a background. You can also assign color codes to variables and use them in the instruction. If you don't specify a color, the current background color will be used. Notice that when you type **CLS** the background color is cleared to the same background color as the rest of the screen.

Setting Up a Border

The final VIEW option, border, allows you to include a border or frame around the viewport. This feature simply enhances the viewports on the screen. It isn't necessary to color the background in addition to framing a viewport, although you may find it helps you to identify a screen area. The border can be any color code allowed for the SCREEN you are using. To put a yellow border around the blue viewport, change line 40 to this:

```
40 VIEW (160,100) - (310,190), 1, 14
```

X,Y Coordinates and Viewports

VIEW permits X,Y coordinates to be manipulated in two ways—as relative or as absolute values. Look again at the last example program. Relative values have been used where coordinate 0,0 is in the upper left corner of the *viewport*, not in the upper left corner of the screen. For instance, to see where point 10,10 is located, type **PSET (10,10)** and press ENTER. Figure 12-3 also shows where the pixel is set.

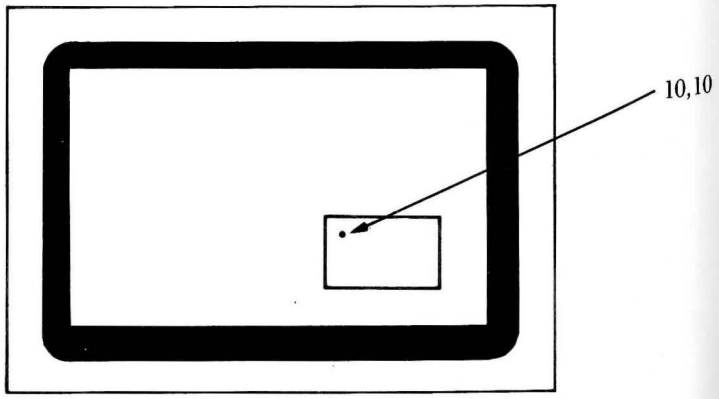


Figure 12-3. *Sample pixel location with VIEW's relative coordinate system*

Even though the upper left of the viewport is 0,0, the lower right diagonal point isn't 319,199. Instead, the lower right corner of a viewport is identified by the difference between the upper left corner of a viewport (*x1,y1*) and the lower right corner of a viewport (*x2,y2*). (In other words, subtract *x2,y2* from *x1,y1*.) In our example, the lower right corner of the viewport will be X = 150 (310 − 160 = 150) and Y = 90 (190 − 100 = 90). So to set a pixel at that point, type **PSET (150,90)** and press ENTER. (It is possible to redefine the entire viewport from 0,0 to 319,199, but that involves using WINDOW.)

The inclusion of VIEW's SCREEN option lets you use the PCjr's absolute X,Y values as coordinates. In this system, the upper left corner is equal to *x1,y1*, while the lower right corner is equal to the coordinate specified by *x2,y2*. Change line 40 in the previous program so that it includes the SCREEN option.

```
40 VIEW SCREEN (160,100) - (310,190), 1,
   14
```

When you run the program, nothing seems to change—a blue viewport is still surrounded by a yellow border. To set a pixel in the upper left of the viewport this time, however, type **PSET (170,110)** and press ENTER. And to set a pixel in the lower right, you might type **PSET (300,180)** and press ENTER. See Figure 12-4.

The following program quickly shows how VIEW can create presentational graphics. In this case, three miniscreens (viewports) like those shown in Figure 12-5 appear on the screen simultaneously. Notice that the three circles are drawn on the screen, each with the same center point (the result of object screen coordinates). Yet because the screen coordinates are redefined three times, the circles are drawn at different locations.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 VIEW (10,10) - (120,75), 9, 4
50 CIRCLE (55,30), 30,, -3.14, -1.57
60 CIRCLE (50,25), 30,, -1.57, -3.14
70 PAINT (60,35), 12, 15: PAINT (45,20),
   12, 15
80 LOCATE 11,5: PRINT "Magazines"
90 VIEW (180,10) - (290,75), 11, 4
100 CIRCLE (55,30), 30,, -4.71, -4.14
110 CIRCLE (53,37), 30,, -4.14, -4.71
```

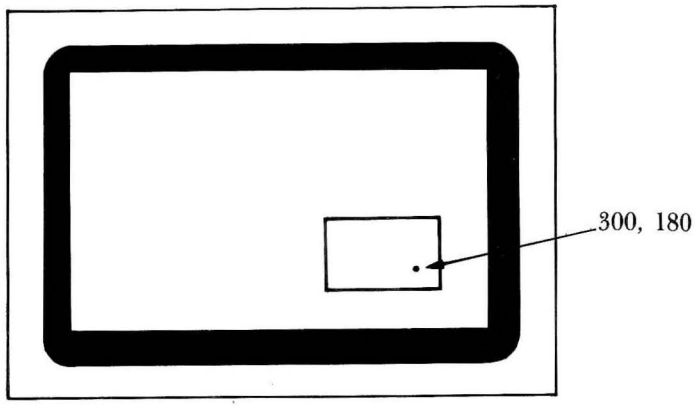


Figure 12-4. *Sample pixel location with VIEW's absolute coordinate system*

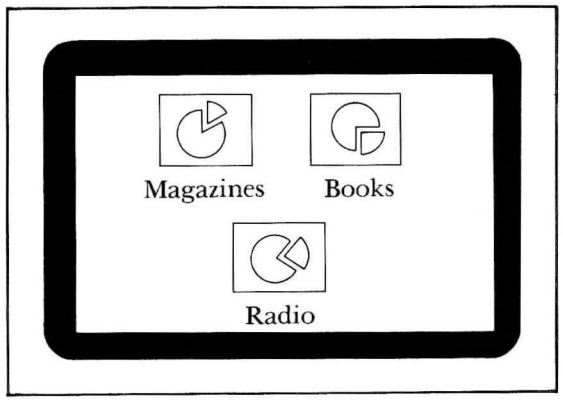


Figure 12-5. *Sample viewports on screen*


```
120 PAINT (60,35), 2, 15: PAINT (50,50),
    2, 15
130 LOCATE 11,28: PRINT "Books"
140 VIEW (105,95) - (205,160), 13, 4
150 CIRCLE (55,30), 30,, -.1, -5.14
160 CIRCLE (60,33), 30,, -5.14, -.1
170 PAINT(45,33),14,15:PAINT (75,45),
    14, 15
180 LOCATE 22,18: PRINT "Radio"
190 END
```

THE WINDOW STATEMENT

WINDOW (described in Syntax 12-2) is associated with the object coordinate system where you can draw an object based upon a coordinate system relative to that object. In this system, the corners of the screen (viewport) are redefined to whatever screen coordinates you specify. When you start up the PCjr, WINDOW is set to the entire screen.

While the syntax of the WINDOW and VIEW statements is similar, the results are different. VIEW redefines the size of the viewing area, and WINDOW controls and redefines the actual screen coordinates within the viewing area. This capability can be quite useful. For example, it allows you to easily translate graphics programs written specifically for computers that use different coordinate systems from those used by the PCjr. Instead of changing the program to fit the PCjr screen coordinates (which might involve writing complex and lengthy functions), you can use WINDOW to define the screen coordinates to match the program.

A side effect of WINDOW is equally useful. This is the effect that lets you "zoom" in on a particular part of an object, then "pan" away from it to regain the large view. Obviously, zooming allows greater detail, while panning provides a bigger, more complete picture. Often you'll want to use WINDOW in conjunction with VIEW, especially with panning or zooming.

When you load BASICA, WINDOW is set to *x1,y1* = 0,0 and *x2,y2* is set to the maximum screen value for the resolution mode you are using. This is 159,199 for low resolution, 319,199 for medium resolution, and 639,199 for high. The purpose of WINDOW, therefore, is to change these values. You can, for instance, have a low-resolution display but with SCREEN coordinates from 0,0 to 639,199. Obviously, this permits a high degree of program flexibility.

WINDOW (*x1,y1*) – (*x2,y2*)
WINDOW SCREEN (*x1,y1*) – (*x2,y2*)

x1 and *y1* specify the upper left corner of a rectangle that represents the new window. *x1* is a numeric expression between:
0 and 159 (low resolution),
0 and 319 (medium resolution),
0 and 639 (high resolution).
y1 is between 0 and 199.

x2 and *y2* specify the lower right corner of a rectangle that represents the new window. *x2* is a numeric expression between:
0 and 159 (low resolution),
0 and 319 (medium resolution),
0 and 639 (high resolution).
y2 is between 0 and 199.

SCREEN is optional; if omitted, the minimum Y-value is at the top of the screen and the maximum at the bottom (that is, the "normal" PCjr screen with 0,0 in the upper left corner). If SCREEN is used, the opposite is true (0,0 in the lower left corner).

Syntax 12-2. *The WINDOW Statement*

PCjr Standard Coordinate System Versus World Coordinate System

In the first part of this book, we mentioned that there are different coordinate systems. Mathematics, for instance, traditionally uses the Cartesian coordinate system, also known as the *world coordinate* system. This system is different from the standard coordinate system the PCjr normally uses. As you know, the PCjr assigns the values 0,0 to the upper left corner of the screen with the maximum value diagonally across the screen in the lower right. The world, or Cartesian system, however, assigns 0,0 to the lower left corner of the screen with the maximum value diagonally across in the upper right. Figure 12-6 illustrates the difference between the two systems.

Unless you include the **SCREEN** option with WINDOW, the PCjr will begin using the world coordinate system. This means that *x1,y1* will be in the lower left corner instead of the upper left, as Figure 12-7 shows.

The following program illustrates the difference between WINDOW with SCREEN and WINDOW without SCREEN.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 WINDOW (0,0)-(319,199)
50 CIRCLE (30,30),10
60 LOCATE 22,10: PRINT "World coordinate
   30,30"
70 FOR DELAY = 1 TO 2000: NEXT DELAY
80 CLS
90 WINDOW SCREEN (0,0)-(319,199)
100 CIRCLE (30,30),10
110 LOCATE 4,10: PRINT "Normal coordinates
    30,30"
120 END
```

In short, be sure to include the SCREEN option if you want to use the normal PCjr coordinate system.

Setting Up the Window

Unlike VIEW or LINE, the diagonal corners of the rectangle you specify with WINDOW can't result in any activity on the screen. Instead, the upper left point (*x1,y1*) simply assigns a value to the upper leftmost point of the current active screen, regardless of the size of that drawing area.

The second parameter, the lower right diagonal point (*x2,y2*), assigns a value to the lower rightmost point on the active drawing area. To change these values, use a WINDOW statement such as **100 WINDOW SCREEN (0,0)–(100,100)**. A program line like this will result in a screen with a coordinate system like the one shown in Figure 12-8.

In the figure, the center of the screen is identified as 50,50. If you did not include the SCREEN option, 50,50 would still be in the center, but the 0,0 and 100,100 values would be located as defined by the world coordinate system.

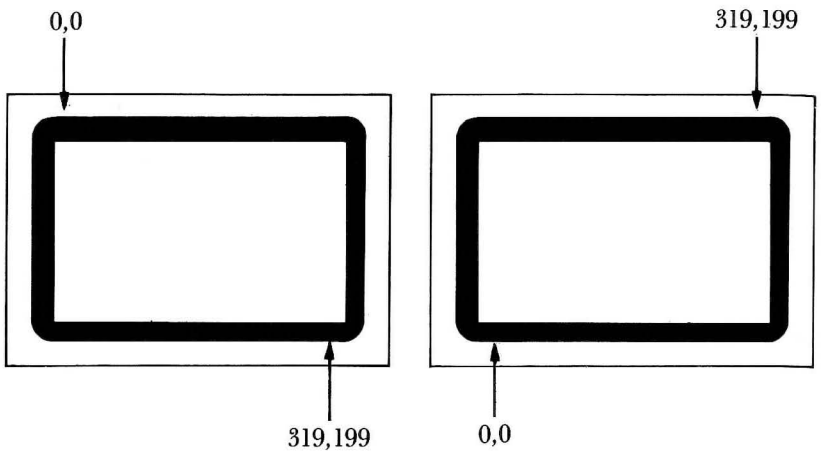


Figure 12-6. *PCjr standard versus world screen coordinates*

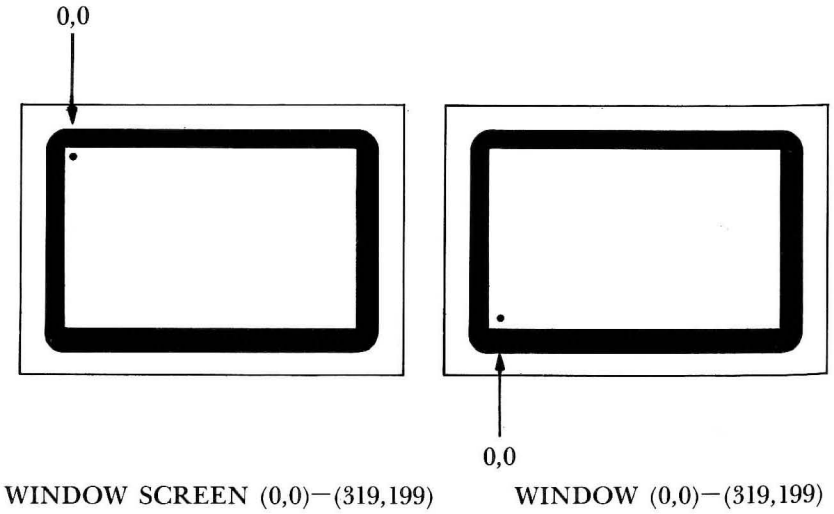


Figure 12-7. *Effect of SCREEN option*

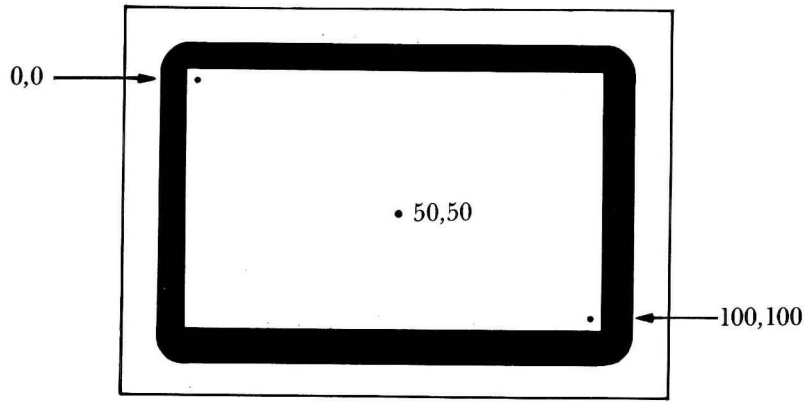


Figure 12-8. *Sample redefined screen coordinates*

As with VIEW, you can specify any valid numeric expression, including variables, to identify X,Y points in a WINDOW statement.

Using WINDOW to Zoom In Or Pan Out on Objects

WINDOW can zoom in on an object, then pan out. Zooming provides more detail, while panning displays more of the object in addition to its surroundings. The next program shows an example of this.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 PI = 3.14159
50 PRINT "Normal view of object..."
60 X = 160: Y = 100: GOSUB 120
70 CLS: PRINT "Zoom in...magnify 5 times
   ..."
80 X = 32: Y = 20: GOSUB 120
90 CLS: PRINT "Pan out...reduce 5 times.
   .."
100 X = 800: Y = 500: GOSUB 120
```

```
110 END
120 WINDOW (-X,-Y) - (X,Y)
130 CIRCLE (0,0), 40, 2, 1.65*PI, 1.35*
    PI, 7/18
140 LINE (-20,10) - (20,-50),1,BF
150 LINE (-10,0) - (10, -40),0
160 LINE (10,0) - (-10,-40),0
170 CIRCLE (0,0), 120, 3, -.25*PI, -.75*
    PI, 2
180 FOR DELAY = 1 TO 2000: NEXT DELAY
190 WINDOW
200 RETURN
```

Using WINDOW With VIEW

The previous discussion of WINDOW has shown how the statement can be used for a variety of applications and effects. When combined with VIEW, however, WINDOW becomes an even more valuable programming tool.

As you saw earlier in this chapter, VIEW can make X,Y coordinates a confusing subject; WINDOW eliminates the confusion by letting you control these coordinates. For instance, VIEW lets you have as many viewports (windows) on the display as you physically have room for. WINDOW lets you assign the "normal" X,Y values to the viewport no matter where it is located. (By normal, we mean 0,0 in the upper left and 319,199 for medium resolution in the lower right.) This makes it an easy matter to duplicate images in a maximum number of viewports with a minimum number of program modifications.

The following program uses PSET to create a sine wave like that shown in Figure 12-9. When you run the program, the normal, full screen is used and the sine wave is drawn across the display.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 LINE (0,100) - (319,100),10
50 X1 = -4 * 3.14159
60 X2 = 4 * 3.14159
70 N = 180: R = 30
80 X = (X2 - X1) / N
90 J = 400 / (X2 - X1)
100 FOR M = X1 TO X2 STEP X
110 X = J * M
```



```
120 Y = R * SIN(M)
130 PSET ((X + 140), (Y + 100)),2
140 NEXT M
150 END
```

After running the program, suppose you decide you want to make it smaller by plotting it inside a viewport. Try adding the following line:

```
35  VIEW (10,110) - (120,170),,4
```

Now when you run the program, you should expect to see the sine wave confined to a box in the lower left corner of the screen—but that doesn't happen. Nothing happens except that a red box appears, followed in a few seconds by the OK prompt. Include VIEW's SCREEN option to see if it helps: **VIEW SCREEN (10,110)–(120,170),,4**. At least when you run the program this time, a green loop appears from the top of the viewport rectangle.

The sine wave doesn't automatically appear inside the viewport because the sine wave is created using a different coordinate range. The active drawing area of the viewport is 10,110 to 120,170. The range for the sine wave is 0,0 to 319,199. You could rewrite the program and base the sine wave on the viewport's coordinates. A much easier way to display the wave, however, is to redefine the window's coordinate system to match the sine wave's by adding line 37 to the previous program.

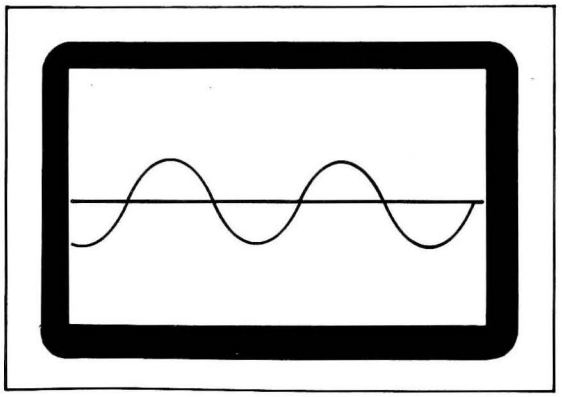


Figure 12-9. *Sample sine wave created with PSET*

The complete program is listed here:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
37 WINDOW SCREEN (0,0) - (319,199)
40 LINE (0,100) - (319,100),10
50 X1 = -4 * 3.14159
60 X2 = 4 * 3.14159
70 N = 180: R = 30
80 X = (X2 - X1) / N
90 J = 400 / (X2 - X1)
100 FOR M = X1 TO X2 STEP X
110 X = J * M
120 Y = R * SIN(M)
130 PSET ((X + 140), (Y + 100)),2
140 NEXT M
150 END
```

Now when you run the program, the screen will look like the one in Figure 12-10.

You might want to return to Chapter 6 and review the bar chart program we used to exemplify the LINE statement. That program created a viewport in the center of the screen, then used WINDOW to

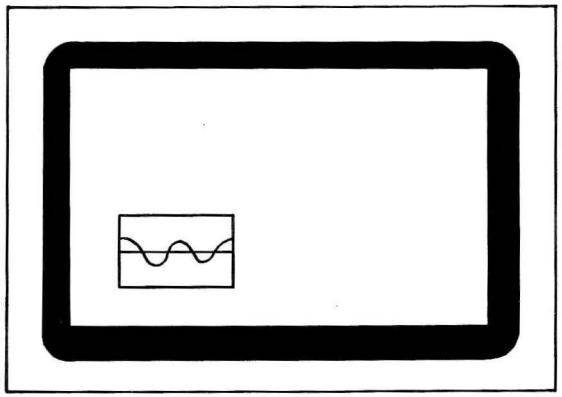


Figure 12-10. *Sample viewport with a sine wave*

begin the coordinate numbering system in the lower left of the window. Notice that the coordinate system extends well beyond the normal coordinates of the PCjr screen. This adds a tremendous amount of program flexibility to any software you may write.

CHAPTER THIRTEEN

Identifying Screen Definitions: POINT and PMAP

There are two PCjr functions that identify screen definitions, including the color of a specific pixel and the current position of the X,Y screen coordinates. The functions POINT and PMAP return these values.

Being able to identify screen definitions is useful for writing and debugging graphics programs in general. It is particularly useful for writing programs that use WINDOW and VIEW, two statements that allow you to redefine the screen. When the screen has been redefined, you may have problems identifying a particular screen location. POINT and PMAP can usually help locate points on the screen.

Note that both POINT and PMAP are called *functions* rather than *statements*. When you enter a statement, something happens on the screen: it changes colors, an object appears, an object disappears. When you enter a function, however, you must use a statement (such as PRINT) before something happens on the screen. This is because functions return values to you. To see that value, you usually must tell the PCjr to print it on the screen or printer.

THE POINT FUNCTION

POINT, which is useful as an error-trapping device or for games that involve boundaries and collisions, has two syntax variations (described in Syntax 13-1).

POINT tests and returns either the color of a specified pixel or the current X or Y location of the *last referenced point (LRP)*. In some cases, you may want the value returned to you; in other cases, you may want the PCjr to take some action based on the value returned. The following program provides one example of how POINT works:

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 PSET (160,100),4
50 A = POINT (160,100)
60 PRINT A
70 END
```

POINT (*x*,*y*)

x specifies a point on the X-axis. It is a numeric expression between:
0 and 159 (low resolution),
0 and 319 (medium resolution),
0 and 639 (high resolution).

y specifies a point on the Y-axis. It is a numeric expression between 0 and 199.

POINT (*n*)

n returns the value of the current X-axis or Y-axis coordinate. If *n* is 0, the current X-value is returned or if *n* is 1, the current Y-value is returned. If WINDOW has previously been used and if *n* is 2, the current world X-value is returned, while if WINDOW has been used and if *n* is 3, the current world Y-value is returned.

Syntax 13-1. *The POINT Function*

When you run this program, the color code 4 will appear on the screen, indicating that the pixel at screen location 160,100 is red.

POINT can be used in more sophisticated programs as well. In the next example, pixels are randomly set in a 20 × 20 grid confined to the upper left corner of the screen. The colors of the pixels are randomly selected, but none of the colors have a color code greater than 5. Using POINT, we can verify when a pixel (10,10) is set to blue (1). In this example, we chose a small grid and a limited number of colors for two reasons. First, the greater the number of points and colors used, the longer it will take for the pixel at 10,10 to turn blue. Second, there is a problem with the RND (random) function that is built into the PCjr's ROM. At times it won't react in a random manner, although it is supposed to. When you run this program, it may end immediately, but it will more likely continue running for at least a few minutes.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 X = 20 * RND
50 Y = 20 * RND
60 C = 5 * RND
70 PSET (X,Y),C
80 IF POINT (10,10) = 1 THEN 200
90 GOTO 40
200 LOCATE 20,1
210 PRINT "10,10 just turned BLUE!"
220 END
```

The second variation of POINT makes it useful for identifying the current screen point position. This variation is especially useful when working with relative moves and the LRP. In Chapter 8, we used DRAW and relative moves to create a three-dimensional cube. DRAW was used to generate the front and back cube faces, while LINE was used for the connecting lines that defined the sides. As you probably discovered when you experimented with the cube, it is easy to use the relative moves to draw the faces, but somewhat more difficult to identify the points to draw the lines to. POINT (*n*) is ideal for identifying the LRP. If at any time you want to know the current X-value of the LRP, type **PRINT POINT (0)** and press ENTER. For the Y-value, type **PRINT POINT (1)**. If you are using WINDOW, use **2** or **3** instead of **0** and **1**. Or you can assign the value returned by POINT to a variable and print it that way. That is what the following program does. Notice

how it uses both forms of POINT (coordinates and color) to display the current X-Y and color values at the bottom of the screen.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 X = 15 * RND
50 Y = 15 * RND
60 C = 5 * RND
70 PSET (X,Y),C
80 X = POINT (0)
90 Y = POINT (1)
100 C = POINT (X,Y)
110 LOCATE 20,1
120 PRINT "X = " X: PRINT "Y = "Y: PRINT
    "The color is " C
130 IF POINT (10,10) = 1 THEN 150
140 GOTO 40
150 PRINT "10,10 just turned BLUE!"
160 END
```

THE PMAP FUNCTION

PMAP is useful as a debugging tool when you are using the WINDOW and VIEW statements because it returns the physical coordinates of a point on the screen. As you know, these two statements allow you to redefine the screen coordinate system. This is sometimes necessary when you want to adapt a program for your PCjr that was written for another computer. Often the other computer's coordinate system is different from the PCjr's, and the program is written specifically for those different coordinates. Normally, the only solution is to calculate complex math functions that translate values from one coordinate system to another. PMAP does this for you. Table 13-1 lists the values to use with the *axis* argument of PMAP (described in Syntax 13-2).

Chapter 12 provided a complete description of screen coordinate definition and redefinition along with definitions of physical and world coordinates. Essentially, physical screen coordinates are the ones the PCjr uses when you first turn the power on (0,0 in the upper left corner, and so on). After you have redefined the coordinate system with WINDOW and VIEW, the new screen coordinate system is referred to as the world coordinate system.

Table 13-1. *PMAP Functions*

PMAP axis	Function
0	Returns the physical screen value of an X-coordinate.
1	Returns the physical screen value of the LRP's Y-coordinate.
2	Returns the world screen value of the LRP's X-coordinate.
3	Returns the world screen value of the LRP's Y-coordinate.

The following program illustrates just what PMAP does. When you run the program, WINDOW redefines the physical coordinates as world coordinates. In this example, 0,0 is assigned to the lower left corner instead of to the upper left. Once the screen has been redefined, you can PSET a pixel by specifying an X- and Y-value when prompted. Finally, the program will set that pixel and tell you the location of the world coordinate in terms of the physical coordinate. If you set pixel

PMAP *(coordinate,axis)*

coordinate specifies the point that is to be mapped. It is a numeric expression between 0 and 639, depending on the resolution mode and on whether the X- or Y-axis is being tested.

axis specifies whether the X- or Y-axis in either the physical or world coordinate system is being mapped. *axis* is a numeric expression between 0 and 3. See Table 13-1 for a description of the values to use.

Syntax 13-2. *The PMAP Function*

10,10, for example, it will be in the lower left instead of the upper left, as shown in Figure 13-1.

```
10 CLEAR ,,,32768
20 SCREEN 5
30 CLS
40 WINDOW (0,0) - (100,100)
50 INPUT "What is the X-value for the
   PSET pixel";X
60 PRINT
70 INPUT "What is the Y-value for the
   PSET pixel";Y
80 PRINT
90 PSET (X,Y)
100 X1 = PMAP(X,0): Y1 = PMAP (Y,1)
110 PRINT "After redefining the screen,
    the world " X "," Y " is at the
    physical " X1 "," Y1
120 END
```

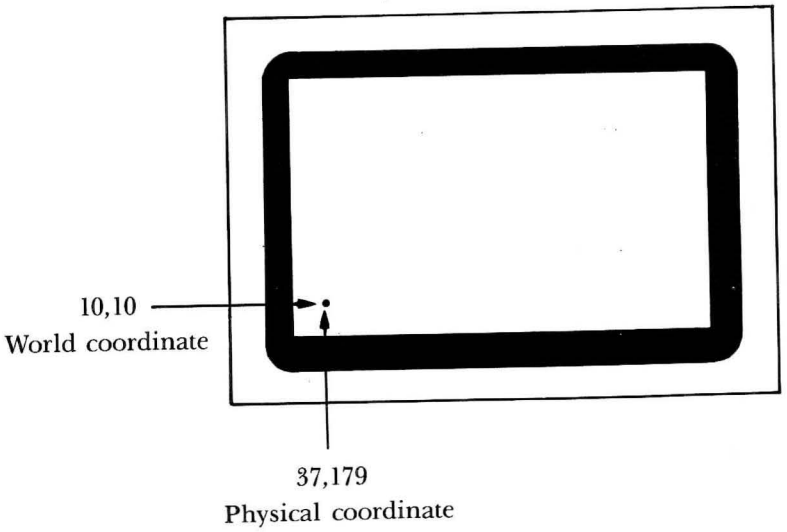


Figure 13-1. *Example of a redefined screen coordinate system*

PART THREE

PCjr Graphics Applications

Once you are familiar with individual graphics commands and functions, you can begin applying them to create vivid screen displays. Part III provides you with many typical graphics applications, including charts, custom characters, and animation techniques.

Each program is divided into modules (or subroutines), and each module is described in the accompanying text. You must type in the *entire program exactly as listed, or errors will result. You can easily adapt each of the programs in this section to your own needs simply by substituting your data for the "dummy" data used to create the images.*

CHAPTER FOURTEEN

Creating Bar Charts And Graphs

Bar graphs are used to display comparisons of related data items. Examples of the kind of information you might display with a bar graph include

- Monthly club attendance for the past year
- Sales by region
- Profit versus gross income for the past five years
- Yearly oil exports by country.

This section demonstrates how to create several types of bar graphs, including simple, grouped, stacked, and three-dimensional charts. There are many other types of bar graphs, but once you know how to write programs for one type, you should be able to create other kinds as well.

Each graph we demonstrate is composed of a main routine and a number of subroutines. You should save each routine after typing it in, so that you will have several complete bar graph programs when you finish this chapter.

SIMPLE BAR CHARTS

The first bar graph program in this section draws a graph of the average monthly class attendance for a twelve-month period. The main routine includes the following lines:

```
1000 'program simple.bar.graph
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1090 'initialize.graphics.
     mode
1040 GOSUB 1240     'draw.ticks
1050 GOSUB 1420     'draw.bars
1060 GOSUB 1630     'draw.titles
1070 GOTO 1070      'wait here forever
1080 '
```

At the heart of most bar graphs (and most other kinds of graphs) is the definition of the origin and extent of the graphics display. To define them, use the VIEW and WINDOW statements. As described in Chapter 12, VIEW defines the physical area (or "viewport") of the screen where graphics commands apply. The WINDOW command defines the corners of the viewport.

Setting Up the Program

Since every good graph has a title, tick labels (measurement marks), and axis labels, we will define the viewport slightly smaller than the actual screen and slightly right of center, so that titles and labels can be printed along the left, top, and bottom.

In the following subroutine an array is dimensioned for GET and PUT activity (line 1150). The array is a little longer than necessary and can always be changed if your requirements change or if you run out of memory. Also note lines 1160–1190. This routine defines variables P.X1, P.X2, P.Y1, and P.Y2 as the "physical" corner points of the viewport, and W.X1, W.X2, W.Y1, and W.Y2 as the logical, or "world," corner points of the viewport. The graph will show 12 bars (for the 12 months of the year), but for spacing purposes we will make the window a little larger than 12. The reason for this should become evident as we proceed.


```
1090'subroutine initialize.graphics.mode
1100 '
1110      CLS
1120      KEY OFF
1130      SCREEN 6
1140      PALETTE
1150      DIM GET.BUFF(50)
1160      P.X1 = 80 : P.Y1 = 16
1170      P.X2 = 635 : P.Y2 = 180
1180      W.X1 = .5 : W.Y1 = 0
1190      W.X2 = 13.2 : W.Y2 = 50
1200      VIEW (P.X1,P.Y1)-(P.X2,P.Y2),,3
1210      WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1220 RETURN
1230 '
```

Drawing the Tick Marks

Tick mark lines, which parallel the X-axis, can now be added to our graph. The following subroutine moves up the Y-axis from the starting world Y-value to the ending world Y-value and draws lines from the starting world X-value to the ending world X-value. You won't draw ticks on the first or last lines, since the viewport lines are already here.

You can select a dotted or dashed line for tick marks instead of a solid line by using the *style* option of the LINE command (line 1270).

Labeling the tick marks requires a bit of manipulation. If you use LOCATE and PRINT, the tick marks probably won't be aligned with the printed rows. To solve this problem, use PRINT to place the label value in the upper corner, put it into a buffer with GET, and then use PUT to align the label with the tick mark.

This method causes a bit of a problem, since BASICA won't permit us to access points outside of the viewport. Fortunately, the PMAP function converts world to physical coordinates, and vice versa, so you can convert a world Y-axis value to a physical value as in line 1280.

PMAP is tricky to use once you've defined a viewport. You will find that in evaluating a value X, PMAP(X,0) returns the value for X as if we were using the default window rather than the viewport X origin. In evaluating a value Y, PMAP(Y,1) returns the value for Y as if we were using the default window and viewport.

Once the physical position for the Y-tick point is derived, we return to the default mapping (lines 1290–1300). We use PRINT and GET to

enter the value for the tick label and then use PUT to line it up three pixels below the tick mark (lines 1310–1340). In line 1340, the label is positioned three pixels below the tick line using PUT. This centers the label with the tick line.

Once the tick labels are drawn (lines 1350–1360), the PCjr returns to the world coordinate system. Lines 1380–1390 clear the last label you printed in the corner.

```
1240 'subroutine draw.ticks
1250 '
1260      FOR TICK=W.Y1 TO W.Y2 STEP 10
1270          IF TICK > W.Y1 AND TICK < W
              .Y2 THEN LINE (W.X1,TICK)-
              (W.X2,TICK)
1280          PHYS.Y = PMAP(TICK,1)
1290          VIEW
1300          WINDOW
1310          LOCATE 1,1
1320          PRINT USING "##";TICK;
1330           GET (0,0)-(15,7),GET.BUFF
1340          PUT (40,PHYS.Y-3),GET.BUFF
1350     VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1360     WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1370      NEXT TICK
1380      LOCATE 1,1
1390      PRINT "  ";
1400 RETURN
1410 '
```

Drawing and Labeling the Bars

Now you can draw in and label the bars. Since this is a demonstration, the following subroutine uses randomly generated data created with the RANDOMIZE statement and RND function (lines 1440 and 1470).

To draw the bar for the data, use the BF (box filled) option of the LINE statement (line 1480). For the X-coordinates, begin the box on the month number (1,2 . . . 12) and make it 0.7 world units wide. The Y-coordinates are simply the low Y-value (W.Y1) and the data value (ATTENDANCE).

You can label the bar with its month at the same time. Again, since the bar may or may not line up with a column from the LOCATE command, save the physical X-value (derived with the PMAP function

in line 1490), return to default mapping, PRINT the value in the upper left corner, GET it, and PUT it where it aligns with the bar (lines 1490–1570).

```
1420 'subroutine draw.bars
1430 '
1440      RANDOMIZE TIMER
1450      MONTH$ = "JanFebMarAprMayJunJul
          AugSepOctNovDec"
1460      FOR MONTH=1 TO 12
1470      ATTENDANCE = 15 + RND(1) * 30
1480           LINE (MONTH,W.Y1)-(MONTH +
               .7,ATTENDANCE),1,BF
1490           PHYS.X = PMAP(MONTH,0) + P.
               X1
1500           VIEW
1510           WINDOW
1520           LOCATE 1,1
1530           PRINT MID$(MONTH$,MONTH * 3
               - 2,3);
1540            GET (0,0)-(23,7),GET.BUFF
1550           PUT (PHYS.X,P.Y2+4),GET.BUFF
1560            VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1570           WINDOW (W.X1,W.Y1)-(W.X2,W.
               Y2)
1580      NEXT MONTH
1590      LOCATE 1,1
1600      PRINT "   ";
1610 RETURN
1620 '
```

Printing the Titles

Now that the graph has been created, you can fill in the main and axes titles. Center the titles by using the following program, which calculates the values for positioning with a simple formula, as shown in lines 1680, 1700, and 1720. (The other method would be to "hardcode" a row/column address with the LOCATE command, but this is generally less flexible.)

Note that the Y-axis requires more work because one letter at a time must be printed, one on top of another (lines 1720–1770). Later in this section you will see how to make fancier titles, writing the Y-axis title with the characters oriented 90 degrees counterclockwise.

```
1630 'subroutine print.titles
1640 '
1650      MAIN.TITLE$ = "A v e r a g e
     C l a s s   A t t e n d a n c e
     " + "   F o r   1 9 8 4"
1660      X.TITLE$ = "Month of the Year"
1670      Y.TITLE$ = "Number of Students
1680      LOCATE 1,44 - LEN(MAIN.TITLE$)
     / 2
1690      PRINT MAIN.TITLE$
1700      LOCATE 25,44 - LEN(X.TITLE$) /
     2
1710      PRINT X.TITLE$;
1720      ROW = 13 - LEN(Y.TITLE$) / 2
1730      FOR CHAR=1 TO LEN(Y.TITLE$)
1740         LOCATE ROW,2
1750          PRINT MID$(Y.TITLE$,CHAR,1);
1760          ROW = ROW + 1
1770      NEXT CHAR
1780 RETURN
1790 '
1800 'End of program
```

Figure 14-1 shows the screen display created by this program.

STACKED BAR CHARTS

A stacked bar chart is used to show how two data items can be combined to display a third data item. For example, consider a graph that shows the wholesale cost and markup price of a grocery item. By stacking the markup on top of the wholesale, you can also show the combined element—the retail price.

This graph will be displayed when you run the next demonstration program. Except for the actual bar drawing, this program is identical to the last one; however, for completeness the entire program is listed later in this section. It shows the wholesale cost, markup, and retail price of a loaf of bread over a 12-month period. The main routine includes these program lines:

```
1000 'program  stacked.bar.graph
1010 '
1020 CLEAR ,,,32768!
```



```
1030 GOSUB 1090    'initialize.
                    graphics.mode
1040 GOSUB 1320    'draw.ticks
1050 GOSUB 1530    'draw.bars
1060 GOSUB 1740    'draw.titles
1070 GOTO 1070     'wait here for <BREAK>
1080 '
```

Setting Up the Program

When you run the entire program, the graph will show a price ranging from 50 cents to 100 cents, which is reflected in the world Y-coordinates (lines 1200–1210). The prices are plotted on a monthly basis over a year's time, so the world X-coordinates will be the same as in the previous graph. Since the graph will have two colors, one for each bar, extra room below the graph will be needed for a color key; consequently, the viewport height is lower (line 1190).

For the sake of variety, we will use red and green for the bar colors (lines 1140–1150). Short tick marks, rather than lines stretching across the graph, are used at each vertical boundary to define the Y-increments. Since this kind of tick mark extends on either side of the boundary (and thus out of the viewport), coordinates (START.X1, START.X2, END.X1, END.X2) must be defined in physical screen terms (lines 1220–1250). **Note:** We define the coordinates as ±2 and ±4. Remember that the boundary line is actually a box just outside of the viewport.

```
1090 'subroutine initialize.graphics.
     mode
1100 '
1110      KEY OFF
1120      SCREEN 6
1130      PALETTE
1140      PALETTE 1,4
1150      PALETTE 2,2
1160      CLS
1170      DIM GET.BUFF(50)
1180      P.X1 = 80 : P.Y1 = 16
1190      P.X2 = 635 : P.Y2 = 172
1200      W.X1 = 0 : W.Y1 = 50
1210      W.X2 = 14 : W.Y2 = 100
1220      START.X1 = P.X1 + 2
1230      END.X1 = P.X1 - 4
```

```
1240      START.X2 = P.X2 - 2
1250      END.X2 = P.X2 + 4
1260      VIEW (P.X1,P.Y1)-(P.X2,P.Y2),,3
1270      WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1280 RETURN
1290 '
```

Drawing the Tick Marks

The subroutine for drawing the tick marks is identical to the previous program, except that instead of a solid line from boundary to boundary, two short lines are drawn on each boundary (lines 1380–1390). Note that since the X-coordinates for these tick marks lie outside of the viewport, we must revert to the default mapping before we can access them (lines 1340–1350).

```
1300 'subroutine draw.ticks
1310 '
1320      FOR TICK = W.Y1 TO W.Y2 STEP 5
1330          PHYS.Y = PMAP(TICK,1)
1340          VIEW
1350          WINDOW
1360          IF TICK = W.Y1 OR TICK=W.Y2
              THEN GOTO 1400
1370              'else
1380                  LINE (START.X1,PHYS
                      .Y)-(END.X1,PHYS.Y)
1390                  LINE (START.X2,PHYS
                      .Y)-(END.X2,PHYS.Y)
1400          LOCATE 1,1
1410          PRINT USING "###";TICK;
1420          GET (0,0)-(23,7),GET.BUFF
1430           PUT (40,PHYS.Y-3),GET.BUFF
1440          VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1450           WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1460      NEXT TICK
1470      LOCATE 1,1
1480      PRINT "   ";
1490 RETURN
1500 '
```


Drawing and Labeling the Bars

Since there are two data items to plot per bar, we will generate two random values, one for the wholesale (a random number between 60 and 90) and a second for the markup (a random number between 0 and 10 added to the wholesale cost).

Once we have the data, the first bar (cost) is drawn with the Y-origin equal to the world Y-origin. The second bar (markup) originates at the top of the cost bar (lines 1580–1590).

The month bar labels are generated exactly as in the previous program (lines 1600–1680).

```
1510 'subroutine draw.bars
1520 '
1530      MONTH$ = "JanFebMarAprMayJunJul
          AugSepOctNovDec"
1540      RANDOMIZE TIMER
1550      FOR MONTH=1 TO 12
1560          COST = 60 + RND(1)*30
1570          MARKUP = COST + RND(1)*10
1580          LINE (MONTH,W.Y1)-(MONTH +
              .7,COST),1,BF
1590          LINE (MONTH,COST)-(MONTH +
              .7,MARKUP),2,BF
1600          PHYS.X = PMAP(MONTH,0) + P.
              X1
1610          VIEW
1620          WINDOW
1630          LOCATE 1,1
1640          PRINT MID$(MONTH$,MONTH * 3
              - 2,3);
1650           GET (0,0)-(23,7),GET.BUFF
1660          PUT (PHYS.X,176),GET.BUFF
1670           VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1680          WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1690      NEXT MONTH
1700      LOCATE 1,1
1710      PRINT "   ";
1720 RETURN
1730 '
```

Printing the Titles

In this program the titles are printed just as they were printed before. However, since there are two colors for bars, you need to state what the colors represent. The simplest way to do this is to draw a colored box next to the key for the color. We'll print the color key text, which is KEY.TITLE$, and draw colored boxes next to the key titles. You could "hardcode" the coordinates for the boxes, but again it is a good idea to use the simple formula to find the start and end of the key line, KEY.TITLE$ (lines 1880–1910).

```
1740 'subroutine print.titles
1750 '
1760     MAIN.TITLE$ = "W h o l e s a l e
         / R e t a i l   C o s t" + "
         F o r   B r e a d - - 1 9 8 4"
1770     X.TITLE$ = "Month of the year"
1780     Y.TITLE$ = "Price in cents"
1790     KEY.TITLE$ = "Wholesale
                       Markup"
1800     LOCATE 1,44 - LEN(MAIN.TITLE$)
         / 2
1810     PRINT MAIN.TITLE$;
1820     LOCATE 24,44 - LEN(X.TITLE$) / 2
1830     PRINT X.TITLE$;
1840     LOCATE 25,44 - LEN(KEY.TITLE$)
         / 2
1850     PRINT KEY.TITLE$;
1860     VIEW
1870     WINDOW
1880     KEY.1.COLUMN = (39 - LEN(KEY.TI
         TLE$) / 2) * 8
1890     KEY.2.COLUMN = (44 + LEN(KEY.TI
         TLE$) / 2) * 8
1900     LINE (KEY.1.COLUMN,191)-(KEY.1.
         COLUMN + 32,199),1,BF
1910     LINE (KEY.2.COLUMN,191)-(KEY.2.
         COLUMN + 32,199),2,BF
1920     ROW = 13 - LEN(Y.TITLE$) / 2
1930     FOR CHAR=1 TO LEN(Y.TITLE$)
1940         LOCATE ROW,2
1950         PRINT MID$(Y.TITLE$,CHAR,1);
1960         ROW = ROW + 1
```



```
1970     NEXT CHAR
1980 RETURN
1990 '
2000 'End of program
```

Figure 14-2 shows the screen display created by this program.

GROUPED BAR CHARTS

Sometimes you may want to display two or more data items that aren't additive as they were with the stacked bar graph. In this case, you will need to "group" the individual bars together under a common label, but separate one member of the group from another with color. This kind of graph is called a *grouped* bar graph.

The demonstration graph in this section displays the gross profits and costs for five sales regions for the month of March. The main routine includes the following program lines:

```
1000 'program grouped.bar.graph
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1090    'initialize.graphics.
                   mode
1040 GOSUB 1300    'draw.ticks
1050 GOSUB 1510    'draw.bars
1060 GOSUB 1740    'print.titles
1070 GOTO 1070     'wait here forever
1080 '
```

Setting Up the Program

Since sales regions rather than months will be plotted, you need to redefine the world X-coordinates so that the five bar groups will line up evenly (lines 1200–1210).

```
1090 'subroutine initialize.
     graphics.mode
1100 '
1110     KEY OFF
```

```
1120      SCREEN 6
1130      PALETTE
1140      PALETTE 1,2
1150      PALETTE 2,4
1160      CLS
1170      DIM GET.BUFF(50)
1180      P.X1 = 80 : P.Y1 = 16
1190      P.X2 = 635 : P.Y2 = 172
1200      W.X1 = .8 : W.Y1 = 1
1210      W.X2 = 6 : W.Y2 = 10
1220      START.X1 = P.X1 - 4
1230      END.X1 = P.X1 + 2
1240      START.X2 = P.X2 - 2
1250      END.X2 = P.X2 + 4
1260      VIEW (P.X1,P.Y1)-(P.X2,P.Y2),,3
1270      WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1280 RETURN
1290 '
```

Drawing the Tick Marks

Drawing the tick marks and labels is done exactly as it was in the previous graphs.

```
1300 'subroutine draw.ticks
1310 '
1320      FOR TICK=W.Y1 TO W.Y2
1330          PHYS.Y = PMAP(TICK,1)
1340          VIEW
1350          WINDOW
1360          IF TICK=W.Y1 OR TICK=W.Y2
              THEN GOTO 1400
1370              'else
1380                  LINE (START.X1,PHYS
                      .Y)-(END.X1,PHYS.Y)
1390                  LINE (START.X2,PHYS
                      .Y)-(END.X2,PHYS.Y)
1400          LOCATE 1,1
1410          PRINT USING "###";TICK;
1420          GET (0,0)-(23,7),GET.BUFF
1430           PUT (40,PHYS.Y-3),GET.BUFF
1440      VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1450           WINDOW (W.X1,W.Y1)-(W.X2,W.
               Y2)
1460      NEXT TICK
```



```
1470      LOCATE 1,1
1480      PRINT "   ";
1490 RETURN
1500 '
```

Drawing the Bars

As in the stacked bar graph, we will generate two random data values per group (lines 1560–1570). Instead of drawing them one on top of the other, however, we draw them side by side, making the second bar (cost) flush with the first (gross) on the horizontal limit instead of the vertical (lines 1580–1590).

```
1510 'subroutine draw.bars
1520 '
1530      REGION$ = "   North      East
             South      West      Central "
1540      RANDOMIZE TIMER
1550      FOR REGION=1 TO 5
1560          GROSS = 5 + RND(1) * 2
1570          COST = 5 + RND(1) * 2
1580          LINE (REGION,W.Y1)-(REGION
              + .4,GROSS),1,BF
1590          LINE (REGION+.4,W.Y1)-(REGI
              ON + .8,COST),2,BF
1600          PHYS.X = PMAP(REGION,0) + P
              .X1
1610          VIEW
1620          WINDOW
1630          LOCATE 1,1
1640          PRINT MID$(REGION$,REGION *
              10-9,10);
1650           GET (0,0)-(79,7),GET.BUFF
1660          PUT (PHYS.X,176),GET.BUFF
1670           VIEW(P.X1,P.Y1)-(P.X2,P.Y2)
1680          WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1690      NEXT REGION
1700      LOCATE 1,1
1710      PRINT "           ";
1720 RETURN
1730 '
```

Printing the Titles

The titles and key are drawn just as they were in the stacked bar graph, changing only the title values.

```
1740 'subroutine print.titles
1750 '
1760      MAIN.TITLE$ = "M a r c h   G r
         o s s   P r o f i t / C o s t
         " + "B y   R e g i o n"
1770      X.TITLE$ = "Sales region"
1780      Y.TITLE$ ="Millions of dollars"
1790      KEY.TITLE$ = "Gross profit
                        Costs"
1800      LOCATE 1,44 - LEN(MAIN.TITLE$)
         / 2
1810      PRINT MAIN.TITLE$;
1820     LOCATE 24,44 - LEN(X.TITLE$) / 2
1830      PRINT X.TITLE$;
1840      LOCATE 25,44 - LEN(KEY.TITLE$)
         / 2
1850      PRINT KEY.TITLE$;
1860     VIEW
1870     WINDOW
1880      KEY.1.COLUMN = (39 - LEN(KEY.TI
         TLE$) / 2) * 8
1890      KEY.2.COLUMN = (44 + LEN(KEY.TI
         TLE$) / 2) * 8
1900      LINE (KEY.1.COLUMN,191)-(KEY.1.
         COLUMN + 32,199),1,BF
1910      LINE (KEY.2.COLUMN,191)-(KEY.2.
         COLUMN + 32,199),2,BF
1920      ROW = 13 - LEN(Y.TITLE$) / 2
1930      FOR CHAR=1 TO LEN(Y.TITLE$)
1940          LOCATE ROW,2
1950 PRINT MID$(Y.TITLE$,CHAR,1);
1960           ROW = ROW + 1
1970      NEXT CHAR
1980 RETURN
1990 '
2000 'End of program
```

Figure 14-3 shows a screen display created by this program.

THREE-DIMENSIONAL BAR CHARTS

To make your bar charts more impressive, you can give the appearance of "depth" to the bars. The following program is identical to the first program except for the way the bars and tick marks are drawn. The main routine looks like this:

```
1000 'program 3.D.bar.graph
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1090     'initialize.
                     graphics.mode
1040 GOSUB 1280     'draw.ticks
1050 GOSUB 1490     'draw.bars
1060 GOSUB 1760     'print.titles
1070 GOTO 1070      'wait here forever
1080 '
```

Setting Up the Program

Since the bars are more complex, we'll use tick marks on the Y-axes as we did in the previous two graphs, rather than solid lines paralleling the X-axis as we did in the first graph (lines 1200—1230).

```
1090 'subroutine initialize.
      graphics.mode
1100 '
1110      KEY OFF
1120      SCREEN 6
1130      PALETTE
1140      CLS
1150      DIM GET.BUFF(50)
1160      P.X1 = 80 : P.Y1 = 16
1170      P.X2 = 635 : P.Y2 = 180
1180      W.X1 = .5 : W.Y1 = 0
1190      W.X2 = 13.5 : W.Y2 = 50
1200      START.X1 = P.X1 - 4
1210      END.X1 = P.X1 + 2
1220      START.X2 = P.X2 - 2
1230      END.X2 = P.X2 + 4
1240      VIEW (P.X1,P.Y1)-(P.X2,P.Y2),,3
```

```
1250      WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1260 RETURN
1270 '
```

Drawing the Tick Marks

The next subroutine draws the tick marks and labels in the same way as in the previous two graphs.

```
1280 'subroutine draw.ticks
1290 '
1300      FOR TICK=W.Y1 TO W.Y2 STEP 10
1310          PHYS.Y = PMAP(TICK,1)
1320          VIEW
1330          WINDOW
1340          IF TICK=W.Y1 OR TICK=W.Y2
              THEN GOTO 1380
1350               'else
1360                    LINE (START.X1,PHYS
                        .Y)-(END.X1,PHYS.Y)
1370                    LINE (START.X2,PHYS
                        .Y)-(END.X2,PHYS.Y)
1380          LOCATE 1,1
1390          PRINT USING "##";TICK;
1400          GET (0,0)-(15,7),GET.BUFF
1410           PUT (40,PHYS.Y-3),GET.BUFF
1420     VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1430           WINDOW (W.X1,W.Y1)-(W.X2,W.
               Y2)
1440      NEXT TICK
1450      LOCATE 1,1
1460      PRINT "  ";
1470 RETURN
1480 '
```

Drawing the Bars

Instead of simply filling a box to draw the bars, the following subroutine draws parallelograms along the top and right of the box and paints the one on the right with a half-tone version of the filled box (lines 1550–1610).


```
1490 'subroutine draw.bars
1500 '
1510      MONTH$ = "JanFebMarAprMayJunJul
          AugSepOctNovDec"
1520      RANDOMIZE TIMER
1530      FOR MONTH=1 TO 12
1540        ATTENDANCE = 10 + RND(1) * 30
1550          LINE (MONTH,W.YO)-(MONTH +
              .5,ATTENDANCE),1,BF
1560          LINE (MONTH,ATTENDANCE)-(MO
              NTH + .4,ATTENDANCE + 2),1
1570          LINE -(MONTH + .9, ATTENDAN
              CE + 2),1
1580          LINE -(MONTH + .5, ATTENDAN
              CE),1
1590          LINE (MONTH + .9, ATTENDANC
              E + 2)-(MONTH + .9, 2),1
1600          LINE -(MONTH + .5,W.YO),1
1610          PAINT (MONTH + .6,5),CHR$(&
              H44)+CHR$(&H11),1
1620          PHYS.X = PMAP(MONTH,0) + P.
              X1
1630          VIEW
1640          WINDOW
1650          LOCATE 1,1
1660          PRINT MID$(MONTH$,MONTH * 3
              -2,3);
1670           GET (0,0)-(23,7),GET.BUFF
1680          PUT (PHYS.X,184),GET.BUFF
1690           VIEW(P.X1,P.Y1)-(P.X2,P.Y2)
1700          WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1710      NEXT MONTH
1720      LOCATE 1,1
1730      PRINT "   ";
1740 RETURN
1750 '
```

Printing the Titles

The graph titles are printed exactly as in the previous bar graphs.

```
1760 'subroutine print.titles
1770 '
1780      MAIN.TITLE$ = "A v e r a g e
          C l a s s   A t t e n d a n c e
          " + "   F o r   1 9 8 4"
```

```
1790      X.TITLE$ = "Month of the Year"
1800      Y.TITLE$ = "Number of Students
1810      LOCATE 1,44 - LEN(MAIN.TITLE$)
          / 2
1820      PRINT MAIN.TITLE$
1830      LOCATE 25,44 - LEN(X.TITLE$)/2
1840      PRINT X.TITLE$;
1850      ROW = 13 - LEN(Y.TITLE$) / 2
1860      FOR CHAR=1 TO LEN(Y.TITLE$)
1870         LOCATE ROW,2
1880          PRINT MID$(Y.TITLE$,CHAR,1);
1890         ROW = ROW + 1
1900      NEXT CHAR
1910 RETURN
1920 '
1930 'End of program
```

Figure 14-4 shows a screen display created by this program.

We've shown you several types of bar graphs, but the variations are limitless. For instance, while in our examples the bars went from bottom to top, making the bars go from left to right or from top to bottom is a simple transition.

In other applications, you may want to display several separate graphs on one screen. To do this, you would simply define several viewports on the screen and draw the bar graphs as previously demonstrated.

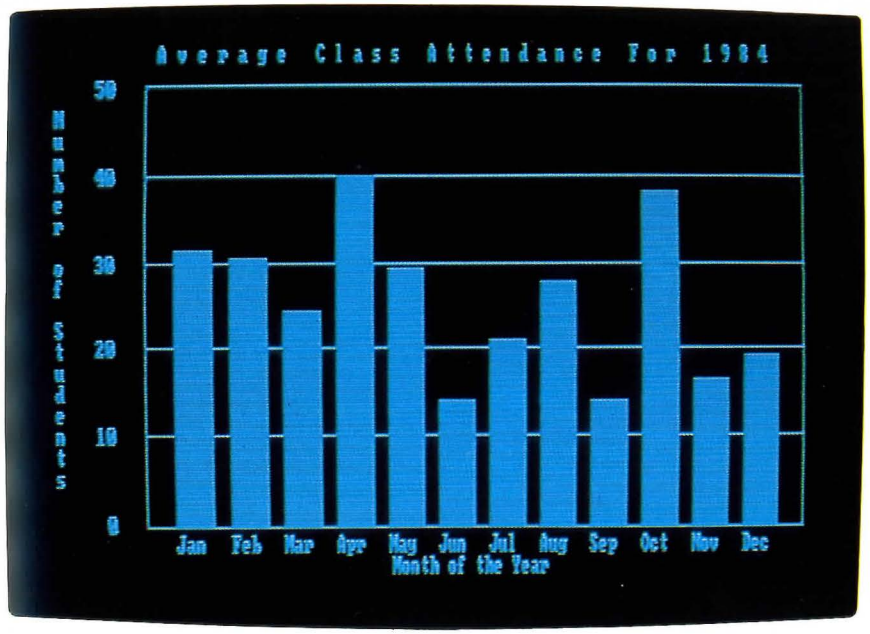


Figure 14-1. *A simple bar chart*

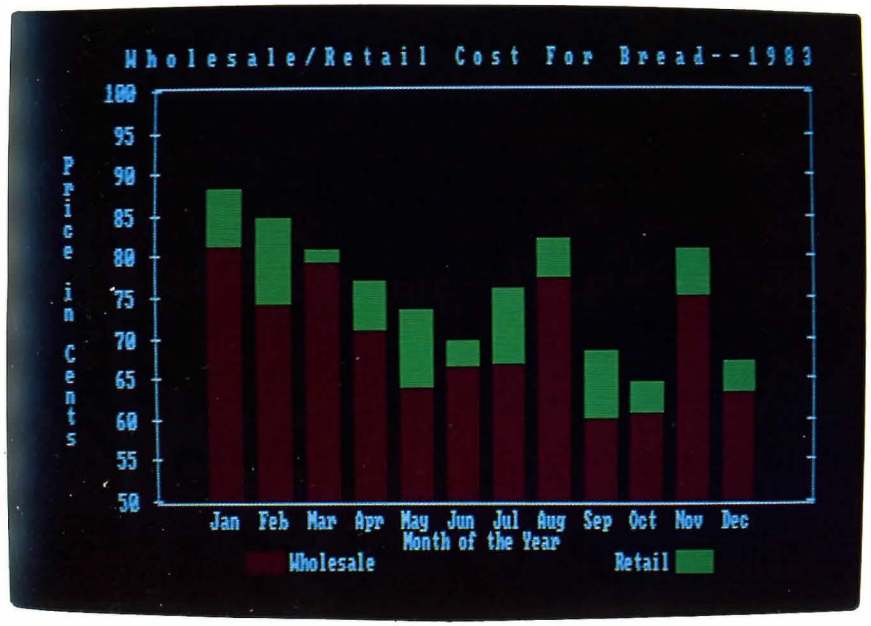


Figure 14-2. *A stacked bar chart*

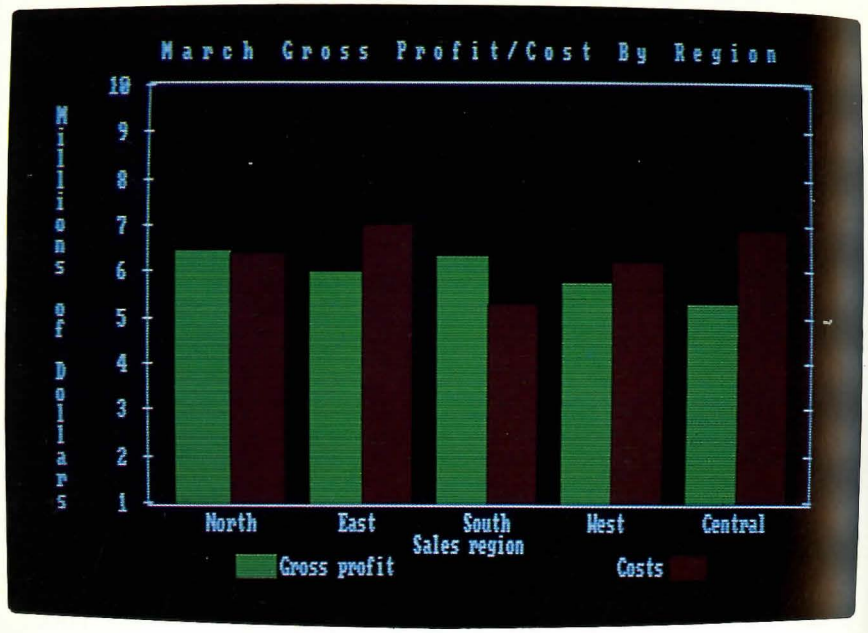


Figure 14-3. *A grouped bar chart*

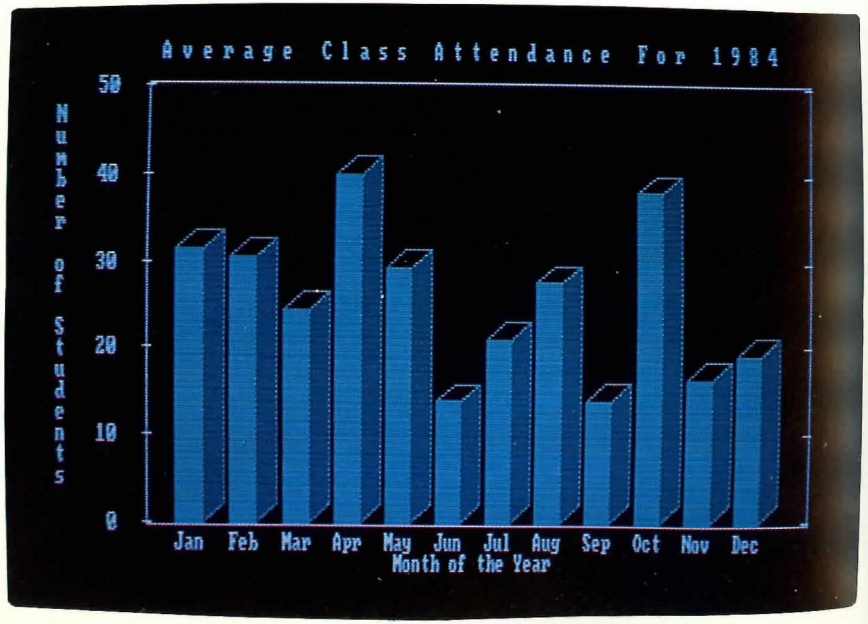


Figure 14-4. *A three-dimensional bar chart*

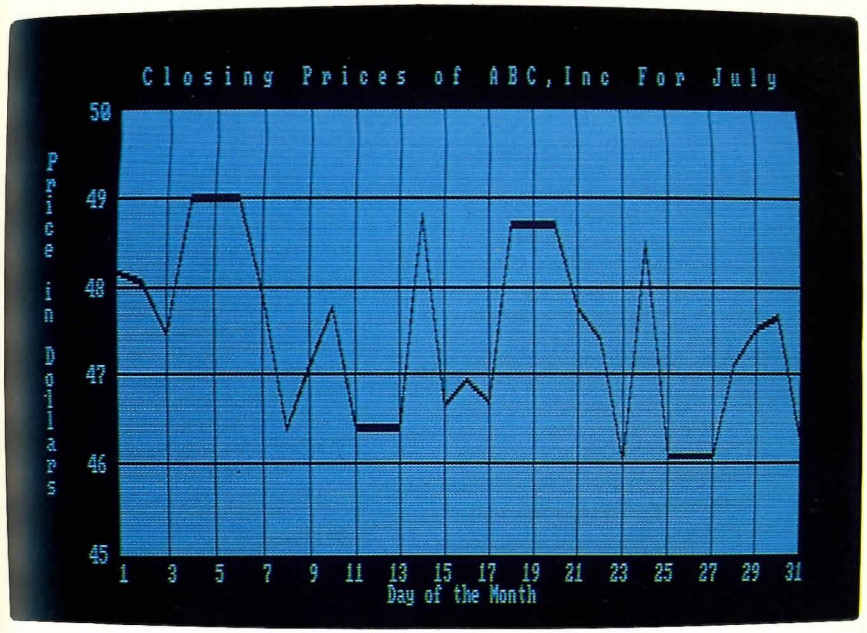


Figure 15-1. *A simple line chart*

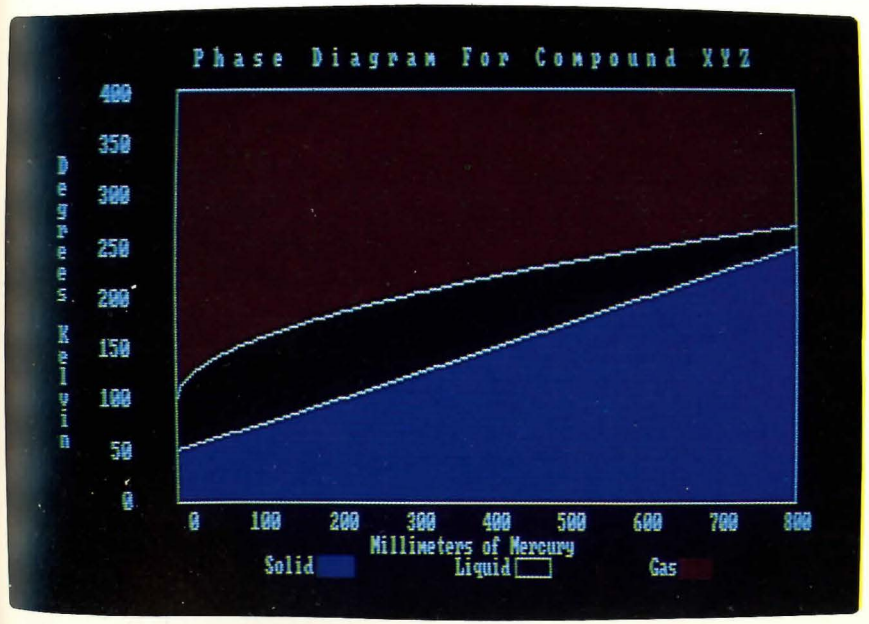


Figure 15-2. *A cumulative line chart*

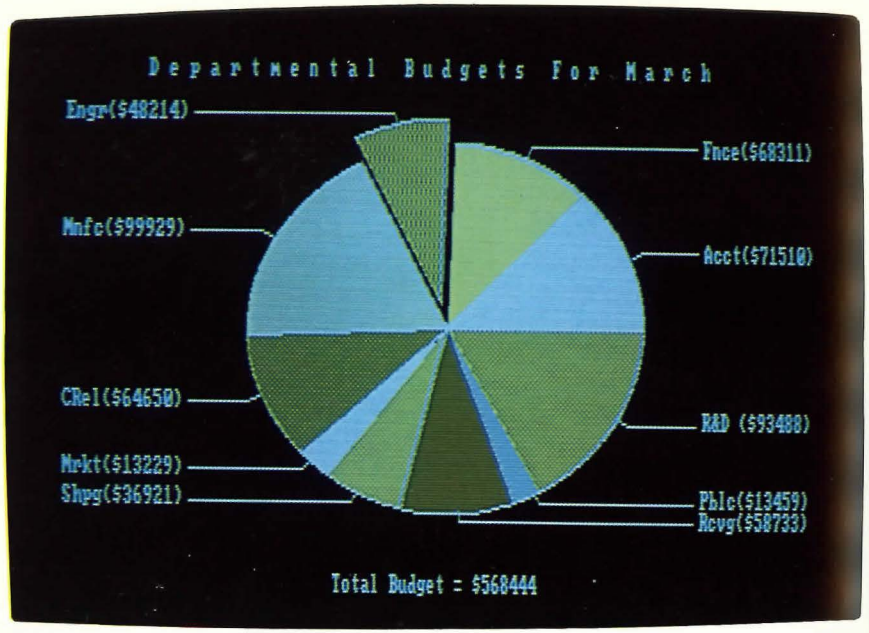


Figure 16-1. *A pie chart with a highlighted slice*

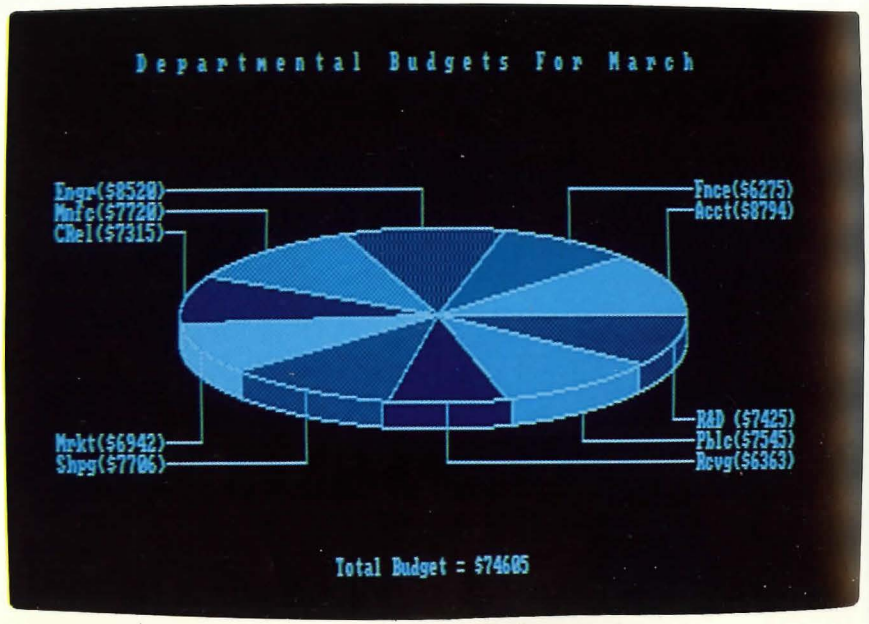


Figure 16-2. *A three-dimensional pie chart*

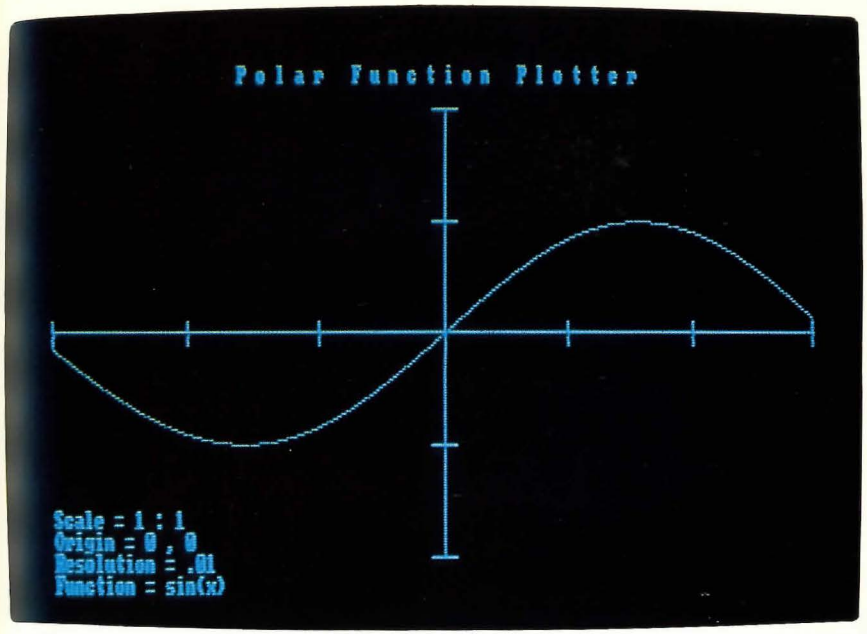


Figure 17-1. *First example of plotted polar coordinates*

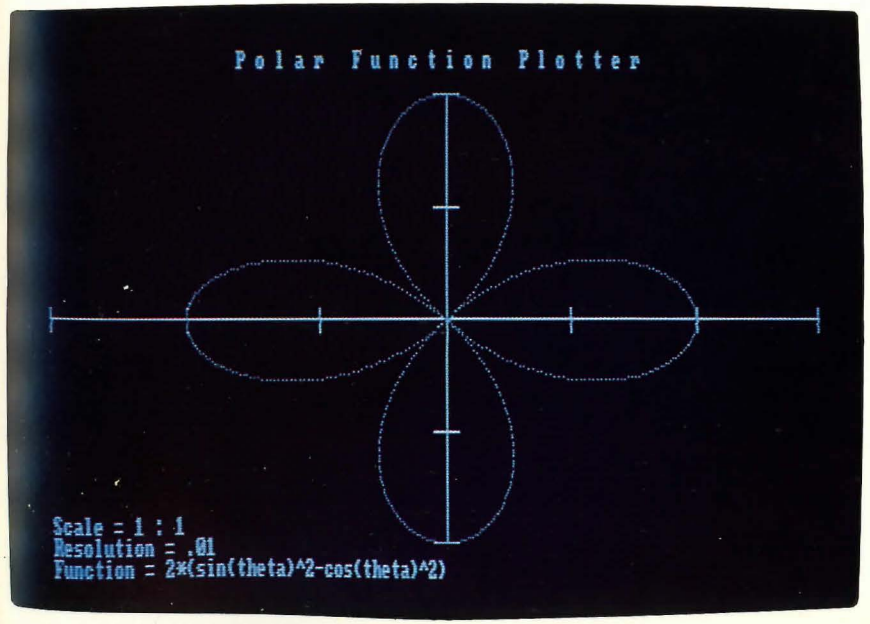


Figure 17-2. *Second example of plotted polar coordinates*

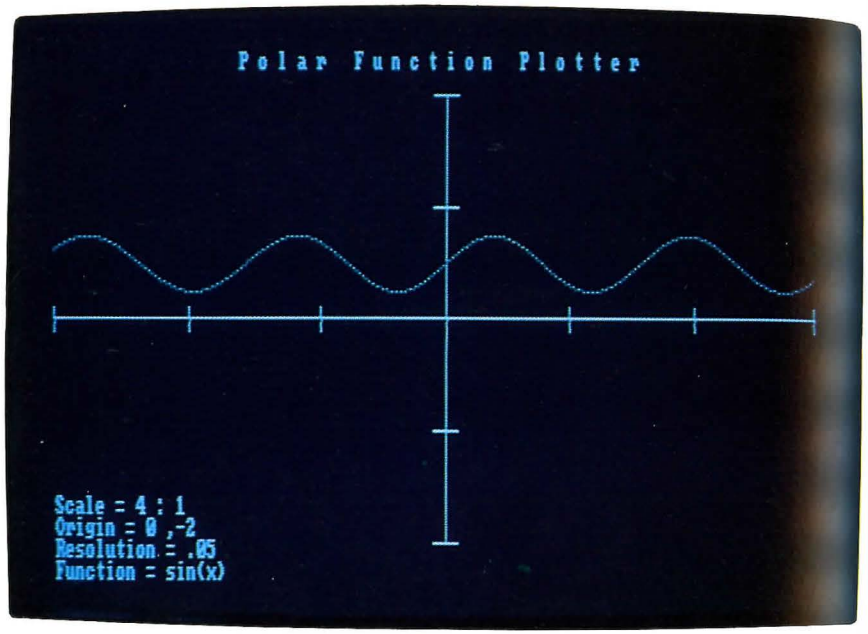


Figure 17-3. *Third example of plotted polar coordinates*

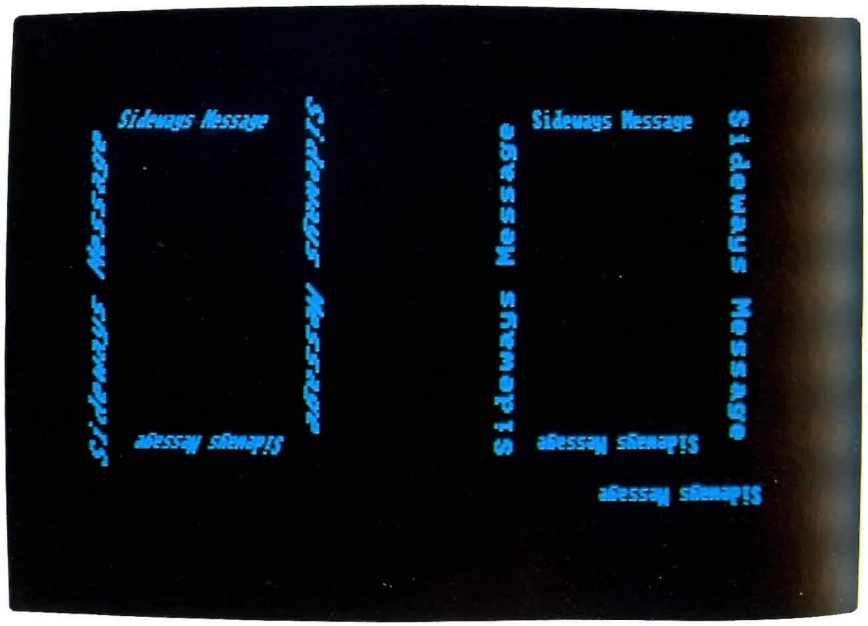


Figure 18-1. *Different writing styles and directions*

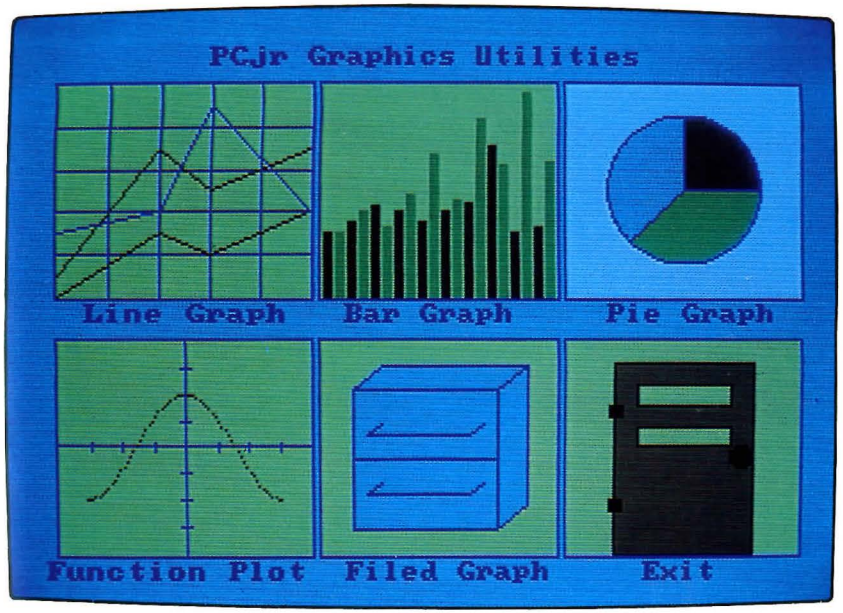


Figure 19-1. *Sample icon menu*

CHAPTER FIFTEEN

Creating Line Charts and Graphs

Line graphs are used to show the effect of one value on another. Everyday examples of line graph applications include

- Plotting stock price trends over a one-month period
- Chemical reaction completion as a function of time
- A car's stopping distance as a function of speed
- A substance's phase (solid, liquid, or gas) as a function of pressure and temperature.

SIMPLE LINE CHARTS

The first line graph demonstration plots stock prices per day over a one-month period. The main routine includes the following programming lines:

```
1000 'program simple.line.graph
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1100      'initialize.
                      graphics.mode
```

```
1040 GOSUB 1260       'generate.dummy.data
1050 GOSUB 1350       'draw.ticks
1060 GOSUB 1660       'draw.lines
1070 GOSUB 1750       'print.titles
1080 GOTO 1080
1090 '
```

Setting Up the Program

As with the bar graph programs, the first step defines the Graphics Mode parameters and the viewport in which drawing occurs. The physical corners of the viewport are defined in the following subroutine, lines 1180–1190. The logical corners of the window are defined in lines 1200–1210. For variety, we will fill in the viewport with white (line 1220).

```
1100 'subroutine initialize.
     graphics.mode
1110 '
1120      KEY OFF
1130      SCREEN 6
1140      PALETTE
1150      PALETTE 1,4
1160      CLS
1170      DIM GET.BUFF(50), CLOSING(31)
1180      P.X1 = 64 : P.Y1 = 16
1190      P.X2 = 630 : P.Y2 = 180
1200      W.X1 = 1 : W.Y1 = 45
1210      W.X2 = 31 : W.Y2 = 50
1220      VIEW (P.X1,P.Y1)-(P.X2,P.Y2),3
1230       WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1240 RETURN
1250 '
```

In our fictitious example, the data consists of the price in dollars of a share of stock in ABC, Inc. The random price varies between 46 and 49 dollars. Since the market is closed on weekends, weekend prices equal the Friday price.

Note: The plotting algorithm requires that the X-values, in this case the days, be in ascending order. Presumably, in a real program you would enter the prices in response to date prompts, which appear in ascending day order.


```
1260 'subroutine generate.dummy.data
1270 '
1280      FOR DAY=W.X1 TO W.X2
1290           IF (DAY MOD 7)=5 OR (DAY MO
               D 7)=6 THEN CLOSING(DAY) =
               CLOSING(DAY-1) : GOTO 1310
1300                'else
1310                     CLOSING(DAY) = 46 +
                         3*RND(1)
1320      NEXT DAY
1330 RETURN
1340 '
```

Drawing the Tick Marks

As with bar graphs, tick marks help show the exact value of the data. Since we are plotting a Y-coordinate as a function of an X-coordinate, it is useful to have vertical and horizontal tick lines. In the next subroutine, lines 1370–1480 draw and label the tick marks for the Y-axis, and lines 1490–1600 draw the tick marks for the X-axis.

```
1350 'subroutine draw.ticks
1360 '
1370      FOR TICK=W.Y1 TO W.Y2
1380           IF TICK > W.Y1 AND TICK < W
               .Y2 THEN LINE (W.X1,TICK) -
               (W.X2,TICK),0
1390           PHYS.Y = PMAP(TICK,1)
1400           VIEW
1410           WINDOW
1420           LOCATE 1,1
1430           PRINT USING "##";TICK;
1440           GET (0,0)-(15,7),GET.BUFF
1450            PUT (40,PHYS.Y-3),GET.BUFF
1460           VIEW(P.X1,P.Y1)-(P.X2,P.Y2)
1470            WINDOW (W.X1,W.Y1)-(W.X2,W.
               Y2)
1480      NEXT TICK
1490      FOR TICK=W.X1 TO W.X2 STEP 2
1500           IF TICK > W.X1 AND TICK < W
               .X2 THEN LINE (TICK,W.Y1) -
               (TICK,W.Y2),0
1510           PHYS.X = PMAP(TICK,0) - 8 +
               P.X1
```

```
1520          VIEW
1530          WINDOW
1540          LOCATE 1,1
1550          PRINT USING "##";TICK;
1560          GET (0,0)-(15,7),GET.BUFF
1570           PUT (PHYS.X,184),GET.BUFF
1580          VIEW(P.X1,P.Y1)-(P.X2,P.Y2)
1590           WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1600      NEXT TICK
1610      LOCATE 1,1
1620      PRINT "  "
1630 '
1640 RETURN
1650 '
```

Drawing the Lines

The algorithm for plotting the lines for the line graph is straightforward. Simply work through the data set, connecting one set of X,Y coordinates with subsequent sets of X,Y coordinates. To make the line stand out, the next subroutine draws the line three times, once using the exact Y-coordinate, once using a value slightly less than the Y-coordinate, and once using a value slightly greater than the Y-coordinate.

```
1660 'subroutine draw.lines
1670 '
1680      FOR DAY=W.X1 TO W.X2-1
1690         LINE (DAY, .02 + CLOSING(DAY
             )) - (DAY + 1, .02 + CLOSING
             (DAY + 1)),1
1700          LINE (DAY,CLOSING(DAY)) -
             (DAY + 1,CLOSING(DAY + 1)),1
1710          LINE (DAY,CLOSING(DAY) -.02)
             - (DAY+1,CLOSING(DAY+1)-
             .02),1
1720      NEXT DAY
1730 RETURN
1740 '
```


Printing the Titles

The next subroutine draws the titles for the graph just as you did for the bar graphs.

```
1750 'subroutine print.titles
1760 '
1770      MAIN.TITLE$ = "C l o s i n g
         P r i c e s   o f   A B C ,
         I n c" + "   F o r   J u l y"
1780      X.TITLE$ = "Day of the Month"
1790      Y.TITLE$ = "Price in Dollars"
1800      LOCATE 1,44 - LEN(MAIN.TITLE$)
         / 2
1810      PRINT MAIN.TITLE$
1820      LOCATE 25,44 - LEN(X.TITLE$)/2
1830      PRINT X.TITLE$;
1840      ROW = 13 - LEN(Y.TITLE$) / 2
1850      FOR CHAR=1 TO LEN(Y.TITLE$)
1860         LOCATE ROW,2
1870          PRINT MID$(Y.TITLE$,CHAR,1);
1880         ROW = ROW + 1
1890      NEXT CHAR
1900 RETURN
1910 '
1920 'End of program
```

Figure 15-1 shows a screen display created by this program.

As you can see from the previous program, it would be simple to add a second line to the graph. To differentiate one line from the other, you could use either different colors or different line styles (lines 1690–1710).

CUMULATIVE LINE CHARTS

In some special cases, you can have a graph with two or more lines that never cross each other, and like the stacked bar graph, the lines may even be cumulative. In such a case you may want to color the area below the first line one color, the area between the first and second lines a second color, and so on.

The next program plots the phase transition line for a substance XYZ as a function of pressure and temperature. (A substance changes phase—between solid, liquid, and gas—at different temperatures depending on the pressure.) Where XYZ is a solid, it will be painted blue; where XYZ is a gas, the area will be red; and where XYZ is a liquid, the area will be left black. The main routine includes the following programming lines:

```
1000 'program painted.line.graph
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1100   'initialize.
                  graphics.mode
1040 GOSUB 1280    'generate.dummy.data
1050 GOSUB 1410    'draw.ticks
1060 GOSUB 1700    'draw.lines
1070 GOSUB 1810    'print.titles
1080 GOTO 1080     'wait here for <BREAK>
1090 '
```

Setting Up the Program

For this program, we will have a set of X- and Y-coordinate pairs for the solid/liquid line (SOLID.LIQUID) and a second set for the liquid/gas line (LIQUID.GAS). In the following subroutine, two arrays are dimensioned to hold the X,Y data; the first element is the pressure (*x*) and the second element is the temperature (*y*).

```
1100 'subroutine initialize.
      graphics.mode
1110 '
1120      KEY OFF
1130      SCREEN 6
1140      PALETTE
1150      PALETTE 1,1
1160      PALETTE 2,4
1170      CLS
1180       P.X1 = 100 : P.Y1 = 16
1190       P.X2 = 620 : P.Y2 = 170
1200       W.X1 = 0 : W.Y1 = 0
1210       W.X2 = 800 : W.Y2 = 400
1220       DATA.SET.COUNT = 30
```



```
1230      DIM GET.BUFF(50),  SOLID.LIQUID(
          DATA.SET.COUNT,1),LIQUID.GAS(DA
          TA.SET.COUNT,1)
1240       VIEW (P.X1,P.Y1)-(P.X2,P.Y2),,3
1250      WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1260 RETURN
1270 '
```

To generate the dummy data, the next subroutine uses pressures from the starting world X-value to the ending world X-value. To generate the temperature values, the routine uses arbitrary functions based on the pressure (lines 1340–1360). In a real situation, your program would probably prompt for pressure and temperature values.

```
1280 'subroutine generate.dummy.data
1290 '
1300      INC = (W.X2-W.X1) / (DATA.SET.
          COUNT-1)
1310      PRESSURE = W.X1
1320      FOR DATA.SET=1 TO DATA.SET.
          COUNT
1330           SOLID.LIQUID(DATA.SET,0) =
               PRESSURE
1340           SOLID.LIQUID(DATA.SET,1) =
               50 + PRESSURE / 4
1350           LIQUID.GAS(DATA.SET,0) =
               PRESSURE
1360           LIQUID.GAS(DATA.SET,1) =
               100 + SQR(PRESSURE) * 6
1370           PRESSURE = PRESSURE + INC
1380      NEXT DATA.SET
1390 RETURN
1400 '
```

Drawing the Tick Labels

Since painting will occur between the lines, it would be inconvenient to put in tick lines, so the following subroutine places labels only along each axis.

```
1410 ' Draw tick marks and tick labels
1420 '
1430      FOR TICK=W.Y1 TO W.Y2 STEP 50
```

```
1440          PHYS.Y = PMAP(TICK,1)
1450          VIEW
1460          WINDOW
1470          LOCATE 1,1
1480          PRINT USING "###";TICK;
1490           GET (0,0)-(15,7),GET.BUFF
1500          PUT (40,PHYS.Y-3),GET.BUFF
1510           VIEW(P.X1,P.Y1)-(P.X2,P.Y2)
1520          WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1530      NEXT TICK
1540      FOR TICK=W.X1 TO W.X2 STEP 100
1550          PHYS.X = PMAP(TICK,0) - 8 +
              P.X1
1560          VIEW
1570          WINDOW
1580          LOCATE 1,1
1590          PRINT USING "###";TICK;
1600          GET (0,0)-(15,7),GET.BUFF
1610           PUT (PHYS.X,175),GET.BUFF
1620     VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1630           WINDOW (W.X1,W.Y1)-(W.X2,W.
              Y2)
1640      NEXT TICK
1650      LOCATE 1,1
1660      PRINT "     ";
1670 RETURN
1680 '
```

Drawing Lines and Painting Areas

The next subroutine connects the X,Y coordinates of one data set with the X,Y coordinates of the subsequent data sets (lines 1710–1740). Once all of the points of the data set are connected, you can locate a spot inside each of the areas to paint the graph (lines 1750–1760).

```
1690 ' Draw and label bars
1700 '
1710      FOR DATA.SET=1 TO DATA.SET.COUN
          T-1
1720          LINE (SOLID.LIQUID(DATA.SET,
              0),SOLID.LIQUID(DATA.SET,1))
              - (SOLID.LIQUID(DATA.SET + 1
              ,0),SOLID.LIQUID(DATA.SET +
              1,1))
```



```
1730          LINE (LIQUID.GAS(DATA.SET,0)
              ,LIQUID.GAS(DATA.SET,1)) - (
              LIQUID.GAS(DATA.SET + 1,0),L
              IQUID.GAS(DATA.SET + 1,1))
1740      NEXT DATA.SET
1750      PAINT (SOLID.LIQUID(1,0),SOLID.
          LIQUID(1,1) - 10),1,3
1760      PAINT (LIQUID.GAS(1,0),LIQUID.
          GAS(1,1) + 10),2,3
1770 RETURN
1780 '
```

Printing the Titles

The graph titles and keys are printed as in previous examples. But since there are three keys to label, there is no easy formula to calculate where to put the key; hence in the following subroutine we "hardcode" in the key color boxes (lines 1990–2010).

```
1790 ' Print graph titles
1800 '
1810      MAIN.TITLE$ = "P h a s e   D i
          a g r a m   F o r   C o m p o u
          n d" + "   X Y Z"
1820      X.TITLE$ = "Millimeters of Merc
          ury"
1830      Y.TITLE$ = "Degrees Kelvin"
1840      KEY.TITLE$ = "Solid
             Liquid                  Gas"
1850      LOCATE 1,44 - LEN(MAIN.TITLE$)
          / 2
1860      PRINT MAIN.TITLE$
1870      LOCATE 24,44 - LEN(X.TITLE$)/2
1880      PRINT X.TITLE$;
1890      ROW = 13 - LEN(Y.TITLE$) / 2
1900      FOR CHAR=1 TO LEN(Y.TITLE$)
1910          LOCATE ROW,2
1920 PRINT MID$(Y.TITLE$,CHAR,1);
1930           ROW = ROW + 1
1940      NEXT CHAR
1950      LOCATE 25,44 - LEN(KEY.TITLE$)
          / 2
1960      PRINT KEY.TITLE$;
1970      VIEW
1980      WINDOW
```

```
1990      LINE (210,192)-(240,199),1,BF
2000     LINE (380,192)-(410,199),3,B
2010      LINE (520,192)-(550,199),2,BF
2020 RETURN
2030 '
2040 'End of program
```

Figure 15-2 shows a screen display created by this program.

Another useful way to use line charts is to fit a line to a set of points. The line may be linear, binomial, or exponential. The formulas required for this calculation are based in linear algebra and are beyond the scope of this book.

CHAPTER SIXTEEN

Creating Pie Charts and Graphs

Pie graphs are used to show how several separate data items (the slices) can be summed to make a whole (the pie). The main point of a pie graph is not to display a specific numeric figure, but rather to show who or what has the biggest or smallest slice of the pie. Specific examples of this include

- A company's budget broken down by department
- Contents of a barrel of oil broken down into its end products
- All registered Democrats in New England grouped by state
- The market share of your company's soap product in relation to the whole soap product market.

Before creating pie charts, however, you need to consider your PCjr monitor (TV or RGB monitor) and how BASICA treats it. Most monitors are wider than they are tall. A typical monitor is about 22cm × 17cm, which is about a 1.3 to 1 ratio of width to height. (In some of its calculations, in fact, BASICA assumes that the average screen has a 4:3 width to height ratio.)

Now recall that the pixel dimensions in SCREEN 6 are 640 × 200. This is about a 3.2 to 1 ratio of width to height. Clearly, then, points are much closer together along the horizontal axis than they are along

the vertical axis. Thus, the following statement draws a rectangular box, taller than it is wide.

```
LINE (0,0)-(100,100),,BF
```

The difference between logical size and physical size shouldn't pose a problem when drawing bars and lines. When you are drawing circles, however, this difference must be taken into account so that you don't draw ellipses instead.

If you have ever studied trigonometry, you may recall that a point on a circle has the following Cartesian coordinates:

$r \cos(theta), r \sin(theta)$

where r is the radius of the circle and *theta* is the angle of the point on the circle from the center.

Consequently, you are able to plot a circle by using PSET to specify that the points for *theta* equal 0–360 degrees. Remember that the COS and SIN functions require radian rather than degree measurement, and that 360 degrees equals 2π radians, which equals 6.28 radians.

The following program should plot a circle with a center point 320,100. When you run this program, however, you'll find it creates an ellipse rather than a circle.

```
100 CLEAR ,,,32768
110 SCREEN 6
120 CLS
130 X.ORIGIN = 320
140 Y.ORIGIN = 100
150 RADIUS = 50
160 FOR THETA=0 TO 6.282 STEP .02
170   X = X.ORIGIN + RADIUS *
    COS(THETA)
180   Y = Y.ORIGIN + RADIUS * SIN(THETA)
190   PSET (X,Y),1
200 NEXT THETA
```

Remember that the CIRCLE statement does not automatically use an "aspect ratio" based on the screen mode and viewport settings. You can confirm this by adding the following line to the program and running it again.

```
210 CIRCLE (X.ORIGIN,Y.ORIGIN),RADIUS
```


As you can see, CIRCLE draws a circle faster and rounder than you could by setting every point on the circle. When drawing pie graphs, you must be able to locate the edge of the circle to separate, label, and paint slices. Since the graph really isn't symmetric in strict physical terms, this can't be done.

Fortunately, the WINDOW command allows you to redefine the corner X,Y coordinates. If you redefine the window so that it has the same 1.3 to 1 ratio as the monitor screen, you should be able to use the PSET method to draw the circle and arrive at the desired result.

While redefining the window, it is a good idea to set the logical center of the screen to the origin (0,0). This eliminates the X.ORIGIN and Y.ORIGIN constants used in the previous program. Also, if the values of the corners are so low that the radius is one unit, you can set RADIUS to 1.

Since a typical monitor screen is 22cm × 17cm, you can define the window from −2.2,−1.7 to 2.2,1.7. Now try the program again.

```
100 CLEAR ,,,32768
110 SCREEN 6
120 CLS
130 WINDOW (-2.2,-1.7)-(2.2,1.7)
140 FOR THETA=0 TO 6.281 STEP .02
150   PSET (COS(THETA),SIN(THETA)),1
160 NEXT THETA
170 CIRCLE (0,0),1,2
```

As you can see, the PSET circle is nearly identical to the CIRCLE version.

SIMPLE PIE CHARTS

Now that the groundwork has been established, we can begin our demonstration program. This program will create a pie chart showing the monthly departmental budgets for the fictitious company ABC, Inc. The main routine includes the following programming lines:

```
1000 'program simple.pie.graph
1010 '
1020 CLEAR ,,,32768!
```

```
1030 GOSUB 1090   'initialize.graphics.
                    mode.and.data
1040 GOSUB 1360   'generate.dummy.data
1050 GOSUB 1480   'draw.graph
1060 GOSUB 1740   'print.title
1070 GOTO 1070    'wait here forever
1080 '
```

Setting Up the Program

The window for this program is set in the following subroutine, from −1.7,−1.3 to 1.7,1.3 (line 1130). This obeys the 1.3:1 ratio rule and produces a circle with a radius of 1 that fills the screen and allows room on all sides for titles and slice labels.

The subroutine also creates a palette to paint the pie in shades of blue. These combinations of blue (color 1), light blue (color 9), and white (color 15) are defined in lines 1140−1180. The PAINT tiles are then defined to these combinations (lines 1210−1300). (You might want to break some of these tiling patterns down into their component colors to reinforce tiling concepts.)

The department names and graph title (lines 1320−1330) are then defined, as well as a constant TWOPI (line 1310), which is naturally two times the value of π.

```
1090 'subroutine initialize.graphics.
      mode.and.data
1100 '
1110 DIM TILE$(10),GET.BUFF(100),
          BUDGET(10)
1120 SCREEN 6
1130 WINDOW (-1.7,-1.3)-(1.7,1.3)
1140 PALETTE 0,0
1150 PALETTE 1,1
1160 PALETTE 2,9
1170 PALETTE 3,15
1180 COLOR 3,0
1190 CLS
1200 KEY OFF
1210  TILE$(1) = CHR$(&HEE)
1220  TILE$(2) = CHR$(&HAA)
1230  TILE$(3) = CHR$(&HA5) + CHR$(&HA5)
     + CHR$(&HA5) + CHR$(&H5A)
1240  TILE$(4) = CHR$(&HAB) + CHR$(&HBA)
1250  TILE$(5) = CHR$(&H95) + CHR$(&H59)
```



```
1260  TILE$(6) = CHR$(&HFB) + CHR$(&HBF)
1270  TILE$(7) = CHR$(&HA9) + CHR$(&H9A)
1280  TILE$(8) = CHR$(&H55)
1290  TILE$(9) = CHR$(&H77)
1300  TILE$(10) = CHR$(&HA5) + CHR$(&H5A)
1310  TWOPI = 2 * 3.14152
1320  DEPARTMENT$ = "AcctFnceEngrMnfcCRel
        MrktShpgRcvgPblcR&D "
1330  TITLE$ = "D e p a r t m e n t a l
     B u d g e t s   F o r  M a r c h "
1340  RETURN
1350  '
```

In the next subroutine, the chart data consists of departmental budgets ranging from $10,000 to $110,000 (line 1410). While generating the budgets, the program keeps track of the total budget (line 1420), which is used later to create the pie slices. The total budget becomes one of the titles on the graph (lines 1440−1450).

```
1360 'subroutine generate.dummy.data
1370 '
1380 RANDOMIZE TIMER
1390 TOTAL.BUDGET = 0
1400 FOR DEPT=1 TO 10
1410   BUDGET(DEPT) = 10000 +
       INT(RND(1) * 100000!)
1420   TOTAL.BUDGET = TOTAL.BUDGET
       + BUDGET(DEPT)
1430 NEXT DEPT
1440 TOTAL.BUDGET$ = STR$(TOTAL.BUDGET)
1450 TOTAL.BUDGET$ = "Total Budget = $"+
     RIGHT$(TOTAL.BUDGET$,LEN(TOTAL.
     BUDGET$)-1)
1460 RETURN
1470 '
```

Drawing the Pie Chart

The next subroutine draws the pie chart. Each slice of the pie is given as a fraction of the total: BUDGET(DEPT)/TOTAL.BUDGET (line 1520). You can convert this dollar value to an angle value by multiplying the fraction times the size of the whole pie, TWOPI. To get the actual angle THETA on the circle, add the fractional angle value to the previous slice angle, LAST.THETA.

Next draw a portion of the circle from LAST.THETA to THETA and create the semicircle using the start/stop options of the CIRCLE statement (line 1530). Note that 0.98 is used as the radius of the circle. We do this to account for arithmetic rounding errors. When calculating the circle perimeter, use the SIN and COS functions.

Once the portion of the circle is drawn, the program calculates the angle halfway between the radial lines of our semicircle in lines 1540–1560. This is done for two reasons: so that you know where to draw the label lines, and so that you have a reference point from which to PAINT the slice.

The slice is painted in line 1570, using the half angle to calculate a point 80% to the edge of the circle. It is painted with the tile pattern defined earlier. Note that if the angle is below 0.1 radians (about six degrees), the slice isn't painted because of rounding errors incurred by BASICA when dealing with such small separations.

Labeling the Pie Slices

The pie slices are labeled in lines 1580–1690. First the program draws a short line away from the circle in line 1580. Then it draws a horizontal line to the point where the label begins (line 1590). Just to be sure that the graph is aesthetically balanced, draw the label line on the right side (if HALF.ANGLE is right of center), or draw the label line on the left side (if it is on the left of center). You can take advantage of the arithmetic sign of the cosine of HALF.ANGLE to decide whether to draw right or left (line 1590).

The label string consists of the department name along with the department budget (lines 1600–1610). As with the line and bar graphs, print the label in a standard location, and then use PUT to position it (lines 1620–1690).

As you may have guessed, this method of labeling the slices isn't foolproof. If you have two small slices adjacent to each other, the label strings may be superimposed on one another. The simplest way to get rid of this problem is to reorder the pie so that the small slices aren't next to each other.

```
1480 'subroutine draw.graph
1490 '
1500 LAST.THETA = 0
1510 FOR DEPT=1 TO 10
```



```
1520    THETA = LAST.THETA + (BUDGET
        (DEPT)/TOTAL.BUDGET) * TWOPI
1530    CIRCLE (0,0),.98,,-LAST.THETA,
        -THETA
1540    HALF.ANGLE = (THETA+LAST.THETA)/2
1550    HALF.X = COS(HALF.ANGLE)
1560    HALF.Y = SIN(HALF.ANGLE)
1570    IF THETA-LAST.THETA > .1 THEN
        PAINT (.8 * HALF.X,.8 * HALF.Y),
        TILE$ (DEPT),3
1580    LINE (HALF.X,HALF.Y)-(HALF.X *
        1.06,HALF.Y * 1.06)
1590    LINE -(SGN(HALF.X) * 1.1,HALF.Y
        * 1.06)
1600    LABEL$ = STR$(BUDGET(DEPT))
1610    LABEL$ = MID$(DEPARTMENT$,DEPT *
        4-3,4) + "($" + MID$(LABEL$,2,LEN
        (LABEL$)-1) + ")"
1620    LOCATE 1,35
1630    PRINT LABEL$;
1640    PHYS.Y = PMAP(HALF.Y * 1.05,1)- 4
1650    WINDOW SCREEN (0,0)-(639,199)
1660    GET (271,0)-(271 + LEN(LABEL$) *
        4-1,7),GET.BUFF
1670    IF HALF.X < 0 THEN PUT (112-LEN
        (LABEL$) * 8,PHYS.Y),GET.BUFF
1680    IF HALF.X >= 0 THEN PUT
        (528,PHYS.Y),GET.BUFF
1690    WINDOW (-1.7,-1.3)-(1.7,1.3)
1700    LAST.THETA = THETA
1710 NEXT DEPT
1720 RETURN
1730 '
```

Printing the Titles

All that remains now is to print the title on the top line of the graph and print out the total budget on the bottom line of the graph.

```
1740 'subroutine print.title
1750 '
1760  LOCATE 1,1
1770  LOCATE 1,40 - LEN(TITLE$) / 2
1780  PRINT TITLE$;
1790  LOCATE 25,40 - LEN(TOTAL.
      BUDGET$) / 2
```

```
1800   PRINT TOTAL.BUDGET$;
1810 RETURN
1820 '
1830 'End of program
```

HIGHLIGHTING A PIE SLICE

If you want to highlight a particular slice, you can do one of two things: either paint it a different color or separate the slice from the rest of the pie. The next demonstration graph shows a pie with one of the slices partially removed. The main routine includes the following programming lines:

```
1000 'program removed.slice.pie.chart
1010 '
1020  CLEAR ,,,32768!
1030  GOSUB 1090   'initialize.graphics.
                   mode.and.data
1040  GOSUB 1360   'generate.dummy.data
1050  GOSUB 1490   'draw.graph
1060  GOSUB 1850   'print.title
1070  GOTO 1070
1080 '
```

Setting Up the Program

The initialization steps in the following subroutine are essentially the same as in the simple pie chart, with the exception of the window size (line 1130). Since one slice will be separated from the rest, you must allow extra space by making the circle a little smaller. The simplest way to do this is to make the window a little larger.

For variety, we will make the color scheme shades of brown, yellow, and white (lines 1140–1180).

```
1090 'subroutine initialize.graphics.
     mode.and.data
1100 '
1110   DIM TILE$(10),GET.BUFF(100)
1120   SCREEN 6
1130   WINDOW (-1.9,-1.4)-(1.9,1.4)
```



```
1140   PALETTE 0,0
1150   PALETTE 1,6
1160   PALETTE 2,14
1170   PALETTE 3,15
1180   COLOR 3,0
1190   CLS
1200   KEY OFF
1210   TILE$(1) = CHR$(&HEE)
1220   TILE$(2) = CHR$(&HAA)
1230   TILE$(3) = CHR$(&HA5) + CHR$(&HA5)
       + CHR$(&HA5) + CHR$(&H5A)
1240   TILE$(4) = CHR$(&HAB) + CHR$(&HBA)
1250   TILE$(5) = CHR$(&H95) + CHR$(&H59)
1260   TILE$(6) = CHR$(&HFB) + CHR$(&HBF)
1270   TILE$(7) = CHR$(&HA9) + CHR$(&H9A)
1280   TILE$(8) = CHR$(&H55)
1290   TILE$(9) = CHR$(&H77)
1300   TILE$(10) = CHR$(&HA5) +
       CHR$(&H5A)
1310   TWOPI = 2 * 3.14152
1320   DEPARTMENT$ = "AcctFnceEngrMnfc
       CRelMrktShpgRcvgPblcR&D "
1330   TITLE$ = "D e p a r t m e n t a l
       B u d g e t s   F o r  M a r c h "
1340 RETURN
1350 '
```

The dummy data is generated exactly as before.

```
1360 'subroutine generate.dummy.data
1370 '
1371  RANDOMIZE TIMER
1380  DIM BUDGET(10)
1390  TOTAL.BUDGET = 0
1400  FOR DEPT=1 TO 10
1410    BUDGET(DEPT) = 10000 +
        INT(RND(1) * 100000!)
1420    TOTAL.BUDGET = TOTAL.BUDGET +
        BUDGET(DEPT)
1430  NEXT DEPT
1440  TOTAL.BUDGET$ = STR$(TOTAL.BUDGET)
1450  TOTAL.BUDGET$ = "Total Budget = $"
      + RIGHT$(TOTAL.BUDGET$,LEN(TOTAL.
      BUDGET$)-1)
1460  REMOVED = 3
1470 RETURN
1480 '
```

Drawing the Pie Chart

This subroutine differs from the last one only in the way that it centers the semicircle and draws the slice label. If the slice is part of the pie, the program uses 0,0 as the center point. The standard HALF.X and HALF.Y coordinates are used to draw the label line (lines 1590–1600). If the slice is to be removed, however, the center point is moved one-eighth of the way out to the edge of the circle (lines 1630–1660). The semicircle is then drawn at that new center.

```
1490 'subroutine draw.graph
1500 '
1510  LAST.THETA = 0
1520  FOR DEPT=1 TO 10
1530    THETA = LAST.THETA + (BUDGET
      (DEPT)/TOTAL.BUDGET) * TWOPI
1540    HALF.ANGLE = (LAST.THETA+THETA)
      / 2
1550    HALF.X = COS(HALF.ANGLE)
1560    HALF.Y = SIN(HALF.ANGLE)
1570    IF DEPT=REMOVED THEN GOTO 1620
1580       'else
1590            X.ORIGIN = 0
1600            Y.ORIGIN = 0
1610            GOTO 1670
1620       'then
1630            X.ORIGIN = .125 * HALF.X
1640            Y.ORIGIN = .125 * HALF.Y
1650            HALF.X = 1.125 * HALF.X
1660            HALF.Y = 1.125 * HALF.Y
1670    CIRCLE (X.ORIGIN,Y.ORIGIN),.98,,
      -LAST.THETA,-THETA
1680    IF THETA-LAST.THETA > .1 THEN
      PAINT (.8 * COS(HALF.ANGLE),
      .8 * SIN(HALF.ANGLE)),TILE$
      (DEPT),3
1690    LINE (HALF.X,HALF.Y)-(HALF.X *
      1.06,HALF.Y * 1.06)
1700    LINE -(SGN(HALF.X) * 1.25,HALF.Y
      * 1.06)
1710    LABEL$ = STR$(BUDGET(DEPT))
1720    LABEL$ = MID$(DEPARTMENT$,DEPT *
      4-3,4) + "($" + MID$(LABEL$,2,
      LEN(LABEL$)-1) + ")"
1730    LOCATE 1,35
1740    PRINT LABEL$;
1750    PHYS.Y = PMAP(HALF.Y * 1.05,1)-4
```



```
1760    WINDOW SCREEN (0,0)-(639,199)
1770    GET (271,0)-(271 + LEN(LABEL$) *
      4-1,7),GET.BUFF
1780    IF HALF.X < 0 THEN PUT (104 -
      LEN (LABEL$) * 8,PHYS.Y),
      GET.BUFF
1790    IF HALF.X >= 0 THEN PUT (536,
      PHYS.Y),GET.BUFF
1800    WINDOW (-1.9,-1.4)-(1.9,1.4)
1810    LAST.THETA = THETA
1820  NEXT DEPT
1830 RETURN
1840 '
```

The titles are printed exactly as before.

```
1850 'subroutine print.title
1860 '
1870    LOCATE 1,40 - LEN(TITLE$) / 2
1880    PRINT TITLE$;
1890    LOCATE 25,40 - LEN(TOTAL.
      BUDGET$) / 2
1900    PRINT TOTAL.BUDGET$;
1910  RETURN
1920 '
1930 'End of program
```

Figure 16-1 shows a screen display created by this program.

THREE-DIMENSIONAL PIE CHARTS

The final pie chart program will make the circle wider than it is high. This creates the illusion that the pie is lying on its side, apparently three-dimensional. Because of computational rounding errors (trying to set more points on the screen than there are pixels), special efforts must be made to keep all of the lines connected and the paint within its boundaries. The main routine includes the following programming lines:

```
1000 'program 3.D.pie.chart
1010 '
1020 CLEAR ,,,32768!
```

```
1030 GOSUB 1090       'initialize.mode.
                      and.data
1040 GOSUB 1410       'generate.dummy.data
1050 GOSUB 1540       'draw.graph
1060 GOSUB 2010       'print.titles
1070 GOTO 1070        'wait here forever
1080 '
```

Setting Up the Program

This graph (line 1130) does not require a perfectly round circle, but one that is wider than it is tall. Therefore, the subroutine does not initialize the window. We define the window to be twice as tall as it is wide. With this configuration, we can use the sine/cosine method to draw a circle with a radius of 1 and it will appear elliptical.

A few more constants are defined: PI (naturally the value of π), CHAR.HEIGHT (the height in pixels of an alphanumeric character), and two Boolean constants, TRUE and FALSE (lines 1310–1360).

```
1090 'subroutine initialize.
     mode.and.data
1100 '
1110  DIM TILE$(10),GET.BUFF(100)
1120  SCREEN 6
1130  WINDOW (-1.5,-3)-(1.5,3)
1140  PALETTE 0,0
1150  PALETTE 1,1
1160  PALETTE 2,9
1170  PALETTE 3,15
1180  COLOR 3,0
1190  CLS
1200  KEY OFF
1210  TILE$(1) = CHR$(&HEE)
1220  TILE$(2) = CHR$(&HAA)
1230  TILE$(3) = CHR$(&HA5) + CHR$(&HA5)
     + CHR$(&HA5) + CHR$(&H5A)
1240  TILE$(4) = CHR$(&HAB) + CHR$(&HBA)
1250  TILE$(5) = CHR$(&H95) + CHR$(&H59)
1260  TILE$(6) = CHR$(&HFB) + CHR$(&HBF)
1270  TILE$(7) = CHR$(&HA9) + CHR$(&H9A)
1280  TILE$(8) = CHR$(&H55)
1290  TILE$(9) = CHR$(&H77)
1300  TILE$(10) = CHR$(&HA5) +
     CHR$(&H5A)
1310  PI = 3.14152
```



```
1320  TWOPI = 2 * PI
1330  CHAR.HEIGHT = 8
1340  CHAR.WIDTH = 8
1350  TRUE = (1=1)
1360  FALSE = (1=0)
1370  DEPARTMENT$ = "AcctFnceEngrMnfc
     CRelMrktShpgRcvgPblcR&D "
1380  TITLE$ = "D e p a r t m e n t a l
     B u d g e t s   F o r   M a r c h"
1390 RETURN
1400 '
```

Dummy data is generated in the next subroutine using essentially the same method as in previous graphs.

```
1410 'subroutine generate.dummy.data
1420 '
1430  RANDOMIZE TIMER
1440  DIM BUDGET(10)
1450  TOTAL.BUDGET = 0
1460  FOR DEPT=1 TO 10
1470    BUDGET(DEPT) = INT(RND(1) *
       5000!) + 5000
1480    TOTAL.BUDGET = TOTAL.BUDGET +
       BUDGET(DEPT)
1490  NEXT DEPT
1500  TOTAL.BUDGET$ = STR$(TOTAL.BUDGET)
1510  TOTAL.BUDGET$ = "Total Budget = $"
     + RIGHT$(TOTAL.BUDGET$,LEN(TOTAL
     .BUDGET$)-1)
1520 RETURN
1530 '
```

Drawing the Pie Chart

This graph differs from the previous two pie charts in several respects. First, an ellipse is used rather than a circle. Since the window is defined in terms of an ellipse, however, BASICA makes perfectly round circles regardless of the window dimensions.

Fortunately, the CIRCLE command has an optional aspect ratio parameter (*ar*). This option allows you to change the ratio of the number of points on the Y-radius to the number of points on the X-radius. (Note that the ratio is always given in terms of the default window dimensions rather than of the current window.) Since BASICA

expects a physical screen size ratio of 1.3:1, and since the default window is 640 × 200, the default aspect is

$$\frac{200 \text{ Y-pixels}}{640 \text{ X-pixels}} \times \frac{1.3 \text{ X-pixels}}{1 \text{ Y-pixel}} = 0.41 \text{ (about } 5/12\text{)}$$

For this graph, we have already decided that our screen ratio should should be 2 Y-pixels to 1 X-pixel. This means that the aspect ratio should be

$$\frac{200 \text{ Y-pixels}}{640 \text{ X-pixels}} \times \frac{1 \text{ X-pixel}}{2 \text{ Y-pixels}} = 0.156$$

To complete the graph, the next subroutine draws two circles, both with the aspect ratio 0.156. The first circle is for the top of the pie (line 1590) and the second is for the bottom of the pie (line 1600). Instead of drawing the top of the pie as in the other two graphs, a series of adjacent semicircles are drawn. However, to ensure that the lines are connected (so that PAINT has boundaries), the entire circle is drawn first, and then semicircles are superimposed on top of it.

To draw the pie bottom, seeing only the front half of it, use PI and TWOPI as the start and stop values. Since the bottom of the pie should be lower than the top, use a point 0.3 world units below origin for its Y-center. You will also need to connect the top and bottom of the pie, which is done in lines 1610–1620.

The top of the pie is drawn exactly like the first pie graph example (lines 1680 and 1760). In addition to the pie top, however, a slice may need to be drawn and painted if the slice faces the front of the pie. Drawing the slice cut is easy—just draw a line straight down from the top of the pie to the bottom of the pie at angle THETA (lines 1720–1740) and at angle LAST.THETA if necessary (lines 1690–1710).

To paint the front of this slice, use a starting point just smaller than THETA along the front edge (line 1750). You must be careful when you decide whether or not to paint the front edge. If THETA is too close to 180 degrees (PI), the starting PAINT point could fall outside of the graph, resulting in improper painting.

Labeling the slices can be a problem. Since the pie is twice as compact as in previous examples, it is impossible to draw the label lines diagonally away from the pie and then to use PUT to position labels so that they match the label lines. The labels would assuredly end up on top of each other.

Instead, use vertical label lines coming out at HALF.ANGLE on the pie slice edge. The first line will go up on the right to a certain height, then move horizontally where the label is positioned with PUT. The second label is one ROW above the first, and so on. If HALF.ANGLE is greater than 90 degrees, move over to the upper left of the pie and work down, and so on (lines 1770–1960).

```
1540 'Draw the graph
1550 '
1560  LAST.THETA = 0
1570  ROW = 7 * CHAR.HEIGHT
1580  CHANGEOVER = FALSE
1590  CIRCLE (0,0),1,,,,.156
1600  CIRCLE (0,-.3),1,,PI,TWOPI,.156
1610 LINE (COS(PI),SIN(PI))-(COS(PI),
     SIN(PI)-.3)
1620  LINE (COS(TWOPI),SIN(TWOPI))-
     (COS(TWOPI),SIN(TWOPI)-.3)
1630  FOR DEPT=1 TO 10
1640  THETA = LAST.THETA + (BUDGET
     (DEPT)/TOTAL.BUDGET) * TWOPI
1650  HALF.ANGLE = (LAST.THETA+THETA)
     / 2
1660  PHYS.X = PMAP(COS(HALF.ANGLE),0)
1670 PHYS.Y = PMAP(SIN(HALF.ANGLE),1)
1680  CIRCLE (0,0),1,,-LAST.THETA,-
     THETA,.155
1690  IF LAST.THETA <= PI + .1 THEN
     GOTO 1720
1700     'else
1710          LINE (COS(LAST.THETA),SIN
             (LAST.THETA)) -
             (COS(LAST.THETA),
             SIN(LAST.THETA)-.3)
1720  IF THETA <= PI + .1 THEN GOTO
     1760
1730     'else
1740          LINE (COS(THETA),SIN
             (THETA))-(COS(THETA),
             SIN(THETA)-.3)
1750          PAINT (COS(THETA-.05),SIN
             (THETA-.05)-.15),TILE$
             (DEPT),3
1760  PAINT (.8 * COS(HALF.ANGLE),.8*
     SIN(HALF.ANGLE)),TILE$(DEPT),3
1770  WINDOW SCREEN (0,0)-(639,199)
1780  LABEL$ = STR$(BUDGET(DEPT))
```

```
1790     LABEL$ = MID$(DEPARTMENT$,DEPT *
         4-3,4)+"($" + MID$(LABEL$,2,LEN
         (LABEL$)-1) + ")"
1800     LOCATE 1,35
1810     PRINT LABEL$;
1820     GET (271,0)-(271 + LEN(LABEL$) *
         4-1,7),GET.BUFF
1830     IF HALF.ANGLE < TWOPI/4 OR HALF.
         ANGLE > 3 * TWOPI / 4 THEN GOTO
         1910
1840        'else
1850             IF HALF.ANGLE >= PI AND
                 NOT CHANGEOVER THEN ROW=
                 17 * CHAR.HEIGHT :
                 CHANGEOVER = TRUE
1860             ROW = ROW + CHAR.HEIGHT
1870             PUT (96-LEN(LABEL$)*8,
                 ROW),GET.BUFF
1880             LINE (PHYS.X,PHYS.Y)-
                 (PHYS.X,ROW+3)
1890             LINE -(96,ROW+3)
1900             GOTO 1960
1910        'then
1920             PUT (542,ROW),GET.BUFF
1930            LINE (PHYS.X,PHYS.Y)-
                 (PHYS.X,ROW+3)
1940             LINE -(540,ROW+3)
1950             ROW = ROW - CHAR.HEIGHT
1960     WINDOW (-1.5,-3)-(1.5,3)
1970     LAST.THETA = THETA
1980   NEXT DEPT
1990 RETURN
2000 '
```

Printing the Titles

The main title and total budget are printed just as in the previous two graphs.

```
2010 'subroutine print.titles
2020 '
2030    LOCATE 1,40 - LEN(TITLE$) / 2
2040    PRINT TITLE$;
2050    LOCATE 25,40 - LEN(TOTAL.BUDGET$)
        / 2
2060    PRINT TOTAL.BUDGET$;
```



```
2070 RETURN
2080 '
2090 'End of program
```

Figure 16-2 shows a sample display of this program.

As with bar and line graphs, there are many other styles of pie graphs. For example, you could leave the pie itself unlabeled and use a color-coded key to the left or right of the pie indicating which each pie represents.

Another form of pie graph may only show part of the pie. To make such a graph, you calculate the angle values as usual, but only draw certain slices.

Still another form of the pie graph may position all of the slices away from the center. To make such a graph, use the procedure given in the second example but apply the procedure to all of the slices.

CHAPTER SEVENTEEN

Creating Function Plots

An interesting academic application of graphics is the plotting of mathematical functions. Often you want to see how a complicated function looks but you don't want to plot it tediously, point by point. Your PCjr, however, can plot the function faster and in greater detail than you could.

Specific examples of function plot applications include

- Plotting trigonometric functions to show their characteristics at certain values.
- Plotting electromagnetic wave functions to show constructive and destructive interference between two waves.
- Plotting electron orbitals of an atom to show their shapes and to show how they bond to orbitals of other atoms.

We will discuss two different ways of plotting functions—by the Cartesian coordinate system and by the polar coordinate system. You are already familiar with the Cartesian system; it is the X,Y coordinate system used by the PCjr to plot points on the screen.

To plot a function of X in the Cartesian system, you must vary the value of X and plot each X-value with the Y-value produced by the function. For example, the following function yields a sine wave when plotted with the Cartesian system:

$$y = \sin(x)$$

SIMPLE FUNCTION PLOTS

The first program will plot a function using the Cartesian coordinate system. The main program includes the following programming lines:

```
1000 'program  cartesian.function.plot
1010 '
1020  CLEAR ,,,32768!
1030  GOSUB  1130 'generate.dummy.data
1040  GOSUB  1260 'initialize.mode.&.data
1050  CHAIN  MERGE "TEMP.DAT",1060,ALL
1060  GOSUB  2000 'define.function
1070  GOSUB  1400 'lines.and.ticks
1080  GOSUB  1570 'print.titles
1090  GOSUB  1670 'plot.function
1100  KILL "TEMP.DAT"
1110 GOTO 1110  'wait here for <BREAK>
1120 '
```

For versatility, the following subroutine uses variables for the X,Y origin as well as for the scale and resolution of the function (lines 1150–1180).

Defining the function provides an interesting problem. As you will recall, BASICA lets you define your own functions with the DEF FN statement. However, unless you want to edit this program every time a function is changed, some means of dynamically adding the DEF FN statement to the program while it is running must be included.

Fortunately, the PCjr's version of BASICA includes the CHAIN statement. This statement allows you, among other things, to merge a program on disk with the program currently in memory. We will simply define our function, write it to the disk in the proper format, and then CHAIN it into the program.

While experimenting with the CHAIN statement, we found a few peculiarities about its use that are not documented in the *PCjr BASIC Reference Manual.* First, the CHAIN statement must be in the main-line code rather than in a subroutine. Second, and more important for our application, CHAIN returns the window and viewport mapping to their default values. Hence the window and viewport must be redefined after CHAIN.

To put the function definition into the correct format, prefix it with a line number and then write it into a temporary file. So that you can execute the statement once it is in the program, write out a RETURN statement so that the program can call the two lines as a subroutine (lines 1210–1220).

Once the function subroutine is written into memory, use CHAIN to bring it into memory (line 1050) and then call the subroutine to define the function (line 1060). After the function is defined, you may use KILL to delete it (line 1100).

```
1130 'subroutine  generate.dummy.data
1140 '
1150      SCALE = 1
1160      RESOLUTION = .01
1170      X.ORIGIN = 0
1180      Y.ORIGIN = 0
1190      FUNCTION$ = "2*(sin(x)^2-
          cos(x)^2)"
1200       OPEN "O",1,"TEMP.DAT"
1210      PRINT #1,"2000 DEF FNX(X)=" +
          FUNCTION$
1220      PRINT #1,"2010 RETURN"
1230      CLOSE 1
1240 RETURN
1250 '
```

Setting Up the Program

So that you can print the graph's title at the top of the screen, set up a viewport slightly smaller than the actual screen size (lines 1320-1330). The graph will plot Y-values within a range of Y.ORIGIN ± two times the SCALE (lines 1360–1370). It will use X-values from X.ORIGIN ± three times the SCALE (lines 1340–1350).

```
1260 'subroutine initialize.mode.&.data
1270 '
1280      SCREEN 6
1290      CLS
1300      KEY OFF
1310      PALETTE
1320       P.X1 = 0 : P.Y1 = 16
1330       P.X2 = 639 : P.Y2 = 185
1340      W.X1 = X.ORIGIN - 3 * SCALE
1350      W.X2 = X.ORIGIN + 3 * SCALE
1360      W.Y1 = Y.ORIGIN - 2 * SCALE
1370      W.Y2 = Y.ORIGIN + 2 * SCALE
1380 RETURN
1390 '
```

Adding the Tick Marks

The plot will have X- and Y-axes lines centered on the screen. The center of the screen where the X- and Y-axes meet will be the point defined as X.CENTER, Y.CENTER. A total of six tick marks will be used on the X-axis and four on the Y-axis.

The numeric value represented by the distance between each tick depends on the SCALE factor that was defined earlier. If SCALE is equal to one, then each tick is one unit long; if SCALE is two, each tick is two units long, and so on. If the origin is 0,0 and the scale is one, then the plot will plot values of X from −3 to +3 into values of Y with a possible range of −2 to +2.

```
1400 'subroutine lines.and.ticks
1410 '
1420      VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1430     WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1440     TICK.SIZE = .1 * SCALE
1450     X.CENTER = (W.X1+W.X2) / 2
1460     Y.CENTER = (W.Y1+W.Y2) / 2
1470     FOR TICK=W.X1 TO W.X2 STEP
         SCALE
1480      LINE (TICK,Y.CENTER-
         TICK.SIZE)-(TICK,Y.CENTER+
         TICK.SIZE)
1490     NEXT TICK
1500     FOR TICK=W.Y1 TO W.Y2 STEP
         SCALE
1510      LINE (X.CENTER-TICK.SIZE,TICK)-
         (X.CENTER+TICK.SIZE,TICK)
1520     NEXT TICK
1530      LINE (W.X1,Y.CENTER)-
         (W.X2,Y.CENTER)
1540      LINE (X.CENTER,W.Y1)-
         (X.CENTER,W.Y2)
1550 RETURN
1560 '
```

Printing the Titles

Since the keys will be displayed in the lower left corner of the plot, the next program prints the key before the function so that the function


```
1570 'subroutine print.titles
1580 '
1590 TITLE$ = "C a r t e s i a n   F u n
     c t i o n   P l o t t e r"
1600      LOCATE 1,40 - LEN(TITLE$) / 2 :
          PRINT TITLE$
1610      LOCATE 22,1 : PRINT "Scale =";
          SCALE;": 1";
1620      LOCATE 23,1 : PRINT "Origin
          =";X.ORIGIN;",";Y.ORIGIN;
1630      LOCATE 24,1 : PRINT "Resolution
          =";RESOLUTION;
1640      LOCATE 25,1 : PRINT "Function =
          ";FUNCTION$;
1650 RETURN
1660 '
```

Plotting the Function

To plot the function, this program simply starts with the lowest value of *x* and moves to the highest value of *x* (line 1690). Next get the functional value of *x* (line 1700). Then use PSET to set the point at the X,Y coordinate.

The program will step through the values of *x* according to the value of RESOLUTION. The smaller the value of RESOLUTION, the more congruent the PSET points will appear, but unfortunately, the slower the plot will execute.

```
1670 'subroutine plot.function
1680 '
1690      FOR X=W.X1 TO W.X2 STEP
          RESOLUTION
1700           Y = FNX(X)
1710           PSET (X,Y),1
1720      NEXT X
1730 RETURN
1740 '
1750 'End of program
```

ZOOMING IN AND PANNING OUT

This is an opportunity to further demonstrate two important concepts in graphics programming that were introduced in Chapter 12—

you imagine that your monitor screen is a camera looking at the object displayed.

If the camera moves closer to the object, the object seems larger; this is called "zooming in." If the camera moves away from the object, the object seems smaller; this is called "zooming out." If the camera moves from left to right or from top to bottom, yet another view of the object is displayed; this is called "panning."

You can simulate zooming and panning by using the BASICA WINDOW statement. If you draw a picture after defining the window **WINDOW (−10,−10)—(10,10)**, the picture comes out in a certain size *n*. If you draw the picture after you define the window as **WINDOW (−20,−20)—(20,20)**, the picture appears smaller (1/2 *n*). In other words, you zoomed out. If you draw the picture after defining the window as **WINDOW (−5,−5)−(5,5)**, the picture appears larger (2 *n*); you zoomed in.

You will recall from the previous program that the magnitude of the X,Y window corners were based partially on the value of SCALE. Increase and decrease the value of SCALE, then rerun the program. You will find that the function zooms in or zooms out, depending on its value.

You can simulate panning by changing the logical center of the graph. If, as before, you define a window **WINDOW (−10,−10)—(10,10)** and then draw an object, the object is displayed based on a 0,0 center. If you then modify either the X-values or the Y-values or both (but keep the difference between the two the same), you can pan the display. For example, if you redefine the window as **WINDOW (0,−10)—(20,10)** and redraw the picture, it is displayed based on a 10,0 center, as though you had moved a camera to the right.

Modify the values of X.ORIGIN and Y.ORIGIN in the previous program and rerun it. You will find that it displays the function in different places based on the value of the origins.

When you study the plot of a function, you might be interested in its behavior at a particular spot on the function, perhaps an intercept or a maximum or minimum. In such a case, it would be helpful to center on that point and zoom in on it. This is a combination of zooming and panning.

To create a graph that uses zoom and pan, make the following changes to the dummy data in the previous program.

```
SCALE = 4
RESOLUTION = .05
Y.ORIGIN = −2
```


PLOTTING POLAR COORDINATES

Plotting a function with the polar coordinate system is a bit different from the type of plotting used in the previous program. Instead of varying the value of X and plotting the Y-values returned by the function, the angle value varies and the tip of the radius produced by the function is plotted. For example, the following function produces a circle when plotted with the polar coordinate system:

$$r = \sin(theta)^2 + \cos(theta)^2$$

Perhaps you thought that BASICA works only in terms of X's and Y's. However, you can plot radii and angles using two very simple formulas for converting radii and angles into X,Y coordinates.

The trigonometric cosine function is defined as

$$\cos(theta) = \frac{x}{\text{radius}}$$

and the sine function is defined as

$$\text{sine}(theta) = \frac{y}{\text{radius}}$$

You can rewrite these equations as

$$x = \text{radius} \times \cos(theta)$$
$$y = \text{radius} \times \sin(theta)$$

giving the values required by the PCjr.

The following program uses the same function as the previous program to illustrate the differences between Cartesian and polar plotting.

```
1000 'program polar.function.plot
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1130 'generate.dummy.data
1040 GOSUB 1240 'initialize.mode.&.data
1050 CHAIN MERGE "TEMP.DAT",1060,ALL
1060 GOSUB 2000 'define function
1070 GOSUB 1380 'print.titles
1080 GOSUB 1470 'lines.and.ticks
1090 GOSUB 1620 'plot.function
1100 KILL "TEMP.DAT"
1110 GOTO 1110
1120 '
```

```
1130 'subroutine generate.dummy.data
1140 '
1150      SCALE = 1
1160      RESOLUTION = .01
1170      FUNCTION$ = "2*(sin(theta)^2-
          cos(theta)^2)"
1180      OPEN "O",1,"TEMP.DAT"
1190      PRINT #1,"2000 DEF FNR(THETA)="
          + FUNCTION$
1200      PRINT #1,"2010 RETURN"
1210      CLOSE 1
1220 RETURN
1230 '
```

Setting Up the Program

The next routine defines the screen as in previous programs, but it does not define an X or Y origin, since formally there are no such constructs in polar plotting. Consider, however, the center of the viewport to be 0,0, and use it when converting from polar coordinates to Cartesian coordinates.

```
1240 'initialize.mode.and.data
1250 '
1260      SCREEN 6
1270      CLS
1280      KEY OFF
1290      PALETTE
1300      P.X1 = 0 : P.Y1 = 16
1310      P.X2 = 639 : P.Y2 = 185
1320      W.X1 = -(3*SCALE)
1330      W.X2 = 3*SCALE
1340      W.Y1 = -(2*SCALE)
1350      W.Y2 = 2*SCALE
1360 RETURN
1370 '
```

Printing the Titles

Next print out the title as before.

```
1380 'subroutine print.titles
1390 '
1400      TITLE$ = "P o l a r   F u n c t
          i o n   P l o t t e r"
```



```
1410      LOCATE 1,40 - LEN(TITLE$) / 2 :
          PRINT TITLE$
1420      LOCATE 23,1 : PRINT "Scale
          =";SCALE;": 1";
1430      LOCATE 24,1 : PRINT "Resolution
          =";RESOLUTION;
1440      LOCATE 25,1 : PRINT "Function
          = ";FUNCTION$;
1450 RETURN
1460 '
```

Although there are no X- or Y-axes in the polar system, in the next program draw a "crosshair" on the center of the screen (lines 1580–1590). The lines of the crosshair define the 0-, 90-, 180-, and 270-degree marks used when plotting the graph. Also draw tick marks to indicate increments of the radius (lines 1510–1570).

```
1470 'subroutine lines.and.ticks
1480 '
1490      VIEW (P.X1,P.Y1)-(P.X2,P.Y2)
1500      WINDOW (W.X1,W.Y1)-(W.X2,W.Y2)
1510      TICK.SIZE = .1 * SCALE
1520      FOR TICK=W.X1 TO W.X2 STEP
          SCALE
1530      LINE (TICK,-TICK.SIZE)-
          (TICK,TICK.SIZE)
1540      NEXT TICK
1550      FOR TICK=W.Y1 TO W.Y2 STEP
          SCALE
1560      LINE (-TICK.SIZE,TICK)-
          (TICK.SIZE,TICK)
1570      NEXT TICK
1580      LINE (W.X1,0)-(W.X2,0)
1590      LINE (0,W.Y1)-(0,W.Y2)
1600 RETURN
1610 '
```

Plotting the Function

Now plot the function. Note that it plots from 0 degrees — 0 radians to 360 degrees — a distance of 6.28 radians (line 1640). The radial values corresponding to each angle (line 1650) are converted into X,Y coordinates (lines 1660–1670). The point is then PSET (line 1680).

```
1620 'subroutine plot.function
1630 '
1640      FOR ANGLE=0 TO 6.28 STEP RESOLUTION
```

```
1650          R = FNR(ANGLE)
1660          X = COS(ANGLE)*R
1670          Y = SIN(ANGLE)*R
1680          PSET (X,Y),1
1690      NEXT ANGLE
1700 RETURN
1710 '
1720 'End of program
```

Figures 17-1, 17-2, and 17-3 show plots of polar functions.

CHAPTER EIGHTEEN

Designing and Displaying Special Characters

Graphics can be used to produce characters in sizes and orientations that cannot be generated with a simple PRINT statement. The style of a character—that is, its shape or design—is referred to as its *font*.

Consider the font your PCjr uses when it displays characters. The PCjr uses an 8 × 8 matrix for each character, regardless of the SCREEN mode. Within this matrix, it uses the first seven pixels for the character and the last pixel as a separator between characters. For most characters, the PCjr uses only the top seven pixels for the character. For *descenders*—letters like p and q with descending tails—the PCjr uses the eighth line.

Using PRINT to display characters is suitable for most applications. Sometimes, however, you need to display characters using a different font—a double- or triple-sized character, for example. The WIDTH statement could be used to change the size of the characters, but you can still only display a single size of character at a time.

When creating graphs, you might want to write a label line sideways, with the characters oriented 90 degrees counterclockwise from normal, so that the label runs from bottom to top or top to bottom. You certainly can't do this with the PRINT statement alone.

Several of the PCjr graphics features help you produce more varied characters and allow you to position characters sideways. BASICA has a function (POINT) that analyzes a particular pixel on the screen and returns its color. It also has a statement, PSET, that sets a particular

pixel to a particular color. With these two statements, you can copy the points on each row of a character to a new character that is oriented in a different direction from the original.

Start by using PRINT to display a message that is to be written sideways. You can then analyze it pixel by pixel and row by row using the POINT function. This allows you to determine which pixels are set and which pixels are not set. As each pixel is tested, a corresponding pixel in the new message can be PSET.

To display characters sideways, you must move either up or down one row on the new message. Similarly, as you move down on the normal message, you must move to the left or right on the sideways message. Figure 18-1 illustrates a program that does this. Run the following program:

```
100 CLEAR ,,,32768!
110 SCREEN 6
120 CLS
130 X.ORIGIN = 100
140 Y.ORIGIN = 100
150 SIDE.X = X.ORIGIN
160 SIDE.Y = Y.ORIGIN
170 MSG$ = "Hello"
180 LOCATE 1,1 : PRINT MSG$
190 FOR Y=0 TO 7
200 FOR X=0 TO LEN(MSG$)*8-1
210 PSET (SIDE.X,SIDE.Y),POINT(X,Y)
220 SIDE.Y = SIDE.Y-1
230 NEXT X
240 SIDE.X = SIDE.X + 1
250 SIDE.Y = Y.ORIGIN
260 NEXT Y
```

The "Hello" message is printed sideways on the screen. However, it does seems to be a bit scanty. Since the SCREEN 6 display is 640 × 200, there are more X-pixels than there are Y-pixels. To compensate for this, each pixel is PSET twice: once at the proper location and once on the pixel to the immediate right. Add the following line to the program:

```
215 PSET (SIDE.X+1,SIDE.Y),POINT(X,Y)
```

and change line 240 to

```
240 SIDE.X = SIDE.X + 2
```


Then run the program again.

The message can be displayed in italics simply by starting each successive row one column less than the previous row. Change line 250 to the following:

```
250 SIDE.Y = Y.ORIGIN + (Y+9) MOD 8
```

PRINTING IN DIFFERENT DIRECTIONS AND IN ITALICS

The following subroutine prints a message left to right, right to left, top to bottom and bottom to top. It can also display italic characters (lines 1340 and 1440).

It requires you to define a starting X-value (SIDE.X), a starting Y-value (SIDE.Y), and a direction (DIR) before you call the routine. DIR has the following definitions:

Start Value		Direction	Font
0	=	left to right	normal
1	=	left to right	italics
2	=	right to left	normal
3	=	right to left	italics
4	=	top to bottom	normal
5	=	top to bottom	italics
6	=	bottom to top	normal
7	=	bottom to top	italics

A short program precedes the subroutine. Each value of DIR calls the subroutine.

```
1000 'program sideways.writing
1010 '
1020 CLEAR ,,,32768!
1030 SCREEN 6
1040 CLS
1050 KEY OFF
1060 MSG$ = "Sideways Message"
```

```
1070 SIDE.X=50 : SIDE.Y=20 : DIR=0 :
     GOSUB 1170
1080 SIDE.X=180 : SIDE.Y=150 : DIR=2 :
     GOSUB 1170
1090 SIDE.X=210 : SIDE.Y=20 : DIR=4 :
     GOSUB 1170
1100 SIDE.X=10 : SIDE.Y=150 : DIR=6 :
     GOSUB 1170
1110 SIDE.X=400 : SIDE.Y=20 : DIR=1 :
     GOSUB 1170
1120 SIDE.X=530 : SIDE.Y=150 : DIR=3 :
     GOSUB 1170
1130 SIDE.X=560 : SIDE.Y=20 : DIR=5 :
     GOSUB 1170
1140 SIDE.X=360 : SIDE.Y=150 : DIR=7 :
     GOSUB 1170
1150 GOTO 1150
1160 '
1170 'Subroutine sideways.writer
1180 '
1190 IF DIR=0 OR DIR=1 THEN Y.INC=1 :
     X.INC=1 : ORIGIN = SIDE.X
1200 IF DIR=2 OR DIR=3 THEN Y.INC=-1 :
     X.INC=-1 : ORIGIN = SIDE.X
1210 IF DIR=4 OR DIR=5 THEN Y.INC=1 :
     X.INC=-2 : ORIGIN = SIDE.Y
1220 IF DIR=6 OR DIR=7 THEN Y.INC=-1 :
     X.INC=2 : ORIGIN = SIDE.Y
1230 LOCATE 1,1
1240 PRINT MSG$;
1250 FOR Y=0 TO 7
1260 IF DIR <= 3 THEN GOTO 1370
1270 'else
1280 FOR X=0 TO LEN(MSG$) * 8-1
1290 PSET (SIDE.X,SIDE.Y),POINT(X,Y)
1300 PSET (SIDE.X+1,SIDE.Y),POINT(X,Y)
1310 SIDE.Y = SIDE.Y + Y.INC
1320 NEXT X
1330 SIDE.Y = ORIGIN
1340 IF (DIR MOD 2)=1 THEN SIDE.Y =
     SIDE.Y - Y.INC*(Y+1)
1350 SIDE.X = SIDE.X + X.INC
1360 GOTO 1450
1370 'then continued
1380 FOR X=0 TO LEN(MSG$)*8-1
1390 PSET (SIDE.X,SIDE.Y),POINT(X,Y)
1400 SIDE.X = SIDE.X + X.INC
1410 NEXT X
1420 SIDE.Y = SIDE.Y + Y.INC
```



```
1430 SIDE.X = ORIGIN
1440 IF (DIR MOD 2)=1 THEN SIDE.X =
     SIDE.X - X.INC*(Y+1)
1450 NEXT Y
1460 RETURN
```

PRINTING BOLDFACE CHARACTERS

Another way to enhance your graphics display is to create boldface characters. The method for boldfacing is quite simple. You use PRINT to display a message, use GET to include the message in an array, and then use PUT to position the message one pixel to the right of where it was originally displayed.

```
100  CLEAR ,,,32768!
110  SCREEN 6
120  DIM GET.BUFF(100)
130  CLS
140  MSG$ = "Boldfaced message
150  ROW = 1
160  GET.Y = (ROW-1) * 8
170  COL = 40 - LEN(MSG$) / 2
180  GET.X = (COL-1) * 8
190  GET.LEN = LEN(MSG$) * 8 / 2
200  LOCATE ROW,COL
210  PRINT MSG$
220  LOCATE ROW+1,COL
230  PRINT MSG$
240  GET (GET.X,GET.Y)-
     (GET.X+GET.LEN,GET.Y+7),GET.BUFF
250  PUT (GET.X+1,GET.Y),GET.BUFF,OR
260  END
```

FORMING CUSTOM CHARACTERS

Now that some of the "brute force" means of producing fancier displays have been illustrated, let's look at a more elegant method. Rather than displaying a phrase and then permutating it with POINT/PSET combinations, you can produce a nonstandard character with a minimum of calculations. There are two ways to do this: using DRAW to form each line and curve in the character, and using a disk file with GET/PUT arrays stored in it.

The first method requires you to put together a string of DRAW commands for each character. This is simple for those characters that are mostly straight lines (the letter T, for example). However, for more complicated characters that have curves (the letter S, for example), you would need a long string of commands to produce a smooth character.

The second option requires you first to create each character and then to save it to a disk file. However, you can utilize programs like the ones just described to do this work for you. In addition, fairly simple programs can be written to handle any editing required to produce a special character.

Creating and Saving Custom Characters

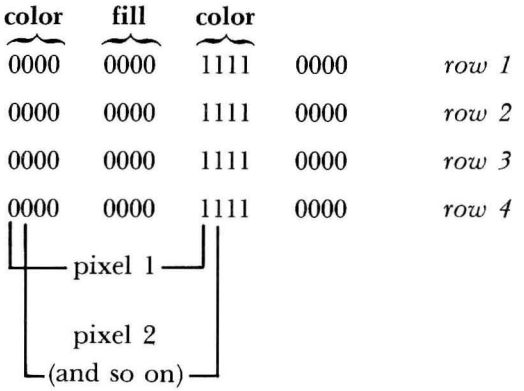
We will describe the second option and use the GET and PUT statements in combination with character sets that will be stored on disk. Remember that BASICA uses different methods to store data generated by GET, depending on the SCREEN mode. Since we have been using screen mode 6 so far, we will only describe the methods BASICA uses for this screen mode. You might find it useful to investigate the other graphics screen modes to contrast the ways that they use GET and PUT.

As Chapter 10 pointed out, each pixel in SCREEN 6 can be one of four colors. Speaking in binary terms, the color can be represented as 00, 01, 10, or 11. When you get an area of the screen (with GET), BASICA stores the sequence of bits in each row. Within each row, it stores all of the first-color bits, and then all of the second-color bits. That is, if a row is made up entirely of color 01, GET stores a string of binary 0s and then a string of binary 1s.

BASICA aligns each row on a byte (eight-bit) boundary. If your GET area is six bits wide, BASICA uses one byte and fills in the last two bits with 0. If your GET area is nine bits wide, BASICA uses two bytes and fills in the last seven bits with 0. After you execute the statements

```
LINE (0,0)-(3,3),1,BF
GET (0,0)-(1,3),GET.ARRAY
```

the array GET.ARRAY contains the data for a 4 × 4 box filled with color 1. The specific bit pattern is as follows:



Along with the actual bit patterns, GET also stores the X,Y dimensions of the array. BASICA allots two bytes of storage in the GET array for each dimension; hence, the preceding example would require four bytes for the dimensions, plus eight bytes for the bit data, or a total of 12 bytes.

We can define a formula for the number of bytes required to store a GET array as follows:

$$\textit{number of bytes} = 4 + \text{INT}\left(\frac{\text{X pixels} + 7}{8}\right) \times 2 \times \text{Y pixels}$$

(Remember that this formula works for SCREEN 6 only!)

The number of array elements required by GET naturally depends on the numeric type of the array. BASICA stores an integer in two bytes, a real number in four, and a double-precision number in eight. Hence, for the above example, you would need either an integer array of six elements, a real array of three elements, or a double-precision array of two elements (actually 1 1/2 elements, but this naturally rounds up to two).

GET puts the X-dimension in the first two bytes of the array, the Y-dimension in the second two bytes, and the bit data in the remainder. If GET.ARRAY (from the previous example) is an integer array, then

GET.ARRAY(0) = 0000 0000 0000 0100 = &H0004

GET.ARRAY(1) = 0000 0000 0000 0100 = &H0004

GET.ARRAY(2) = 0000 0000 1111 0000 = &H00F0

GET.ARRAY(3) = 0000 0000 1111 0000 = &H00F0

GET.ARRAY(4) = 0000 0000 1111 0000 = &H00F0

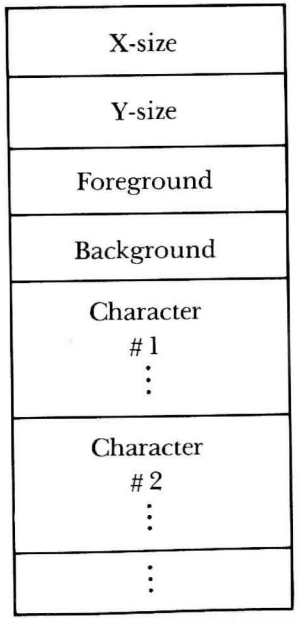
GET.ARRAY(5) = 0000 0000 1111 0000 = &H00F0

Because of their internal format, you can't read the values of real and double-precision array elements as easily as you can read integer arrays. If you could, you would find the same bit patterns as described earlier.

Determining the Disk Files for the Custom Characters

The disk files that store the custom characters are random access, with 2 bytes per record. Each file will start off with a "header" made up of the X-size (record 0), the Y-size (record 1), the foreground color (record 2), and the background color (record 3).

Following the header comes the bit data. Since integer arrays are the easiest to work with, use them for the GET arrays. You already have the X and Y sizes, so only store the actual bit data from the GET array. Each character is grouped in a block of records equal to the number of elements in the array minus the two elements used for the X,Y dimensions as in the following:



Since you will often be using actual ASCII characters for the data, each file will have enough records to accommodate 128 characters (ASCII codes 1–128, for example). The following program generates a file (called "FONT1.DAT") made up of white ASCII characters on a black background, turned 90 degrees counterclockwise. You will recognize the formula for creating these characters from the previous program, "sideways.writer."

Setting Up the Character File

The following program defines the mode and opens the file:

```
1000 'program character.generator
1010 '
1020 CLEAR ,,,32768!
1030 SCREEN 6
1040 CLS
1050 DEFINT A-Z
1060 DIM GET.BUFF(100)
1070 OPEN "R",1,"FONT1.DAT",2
1080 FIELD 1,2 AS CHAR$
```

The next program builds the file header by inserting the X and Y dimensions as well as the foreground color (3) and the background color (0).

```
1090 LSET CHAR$=MKI$(16)
1100 PUT 1,1
1110 LSET CHAR$=MKI$(8)
1120 PUT 1,2
1130 LSET CHAR$=MKI$(3)
1140 PUT 1,3
1150 LSET CHAR$=MKI$(0)
1160 PUT 1,4
```

The next program rotates the characters 90 degrees and copies the bit patterns to disk. For now, duplicate only ASCII codes 32 through 128. You can skip codes 1–31 since these are usually used for "control" characters. In other applications, you may choose to use these codes for special graphics characters.

```
1170 FOR I=32 TO 128
1180 LOCATE 1,1 : PRINT CHR$(I)
1190 FOR X=0 TO 7
1200 FOR Y=0 TO 7
1210 PSET (100+Y*2,107-X),POINT(X,Y)
1220 PSET (101+Y*2,107-X),POINT(X,Y)
1230 NEXT Y
1240 NEXT X
1250 GET (100,100)-(107,107),GET.BUFF
1260 BASE = (I-1)*16+5
1270 FOR REC=2 TO 17
1280 LSET CHAR$=MKI$(GET.BUFF(REC))
1290 PUT 1,BASE+REC-2
1300 NEXT REC
1310 NEXT I
1320 CLOSE
1330 END
```

Using the Character File

The following program accesses the previously created character file. First, it defines the mode and opens the character file.

```
1000 'program vertical.characters
1010 '
1020 CLEAR ,,,32768!
1030 SCREEN 6
1040 CLS
1050 DEFINT A-Z
1060 DIM GET.BUFF(100)
1070 OPEN "R",1,"FONT1.DAT",2
1080 FIELD 1,2 AS CHAR$
```

The next program uses GET to get the dimension values from the file header and calculate the size (in array elements) of each character.

```
1090 GET 1,1 : GET.BUFF(0) = CVI(CHAR$)
1100 GET 1,2 : GET.BUFF(1) = CVI(CHAR$)
1110 FT.SIZE = INT((GET.BUFF(0)+7)/8)*
     GET.BUFF(1)
```

Now you can define the title phrase and call the subroutine to display it in the following program.


```
1120 FT.PHRASE$ = "Vertical Characters"
1130 FT.X = 10 : FT.Y = 92 +
     LEN(FT.PHRASE$)*GET.BUFF(1)/2
1140 GOSUB 1280          'put.font.phrase
```

At this point, you can display the characters stored in the file. Note how line 1220 in the next program defines the X value: multiplying X by 2 double-spaces each line.

```
1150 ASCII = 32
1160 FOR ROW=0 TO 3
1170 FT.PHRASE$ = ""
1180 WHILE LEN(FT.PHRASE$)<24
1190 FT.PHRASE$ = FT.PHRASE$ +
     CHR$(ASCII)
1200 ASCII = ASCII + 1
1210 WEND
1220 FT.X = FT.X + GET.BUFF(0)*2
1230 FT.Y = 92 +
     LEN(FT.PHRASE$)*GET.BUFF(1)/2
1240 GOSUB 1280      'put.font.phrase
1250 NEXT ROW
1260 GOTO 1260
1270 '
```

The next subroutine reads in and displays a character. It essentially reverses the routine used to store the character.

```
1280 'subroutine put.font.phrase
1290 '
1300 FOR C=1 TO LEN(FT.PHRASE$)
1310 CODE = ASC(MID$(FT.PHRASE$,
     C,1))
1320 BASE = (CODE-1)*FT.SIZE+4
1330 FOR REC=1 TO FT.SIZE
1340 GET 1,BASE+REC
1350 GET.BUFF(REC+1) = CVI(CHAR$)
1360 NEXT REC
1370 PUT (FT.X,FT.Y),GET.BUFF
1380 FT.Y = FT.Y - GET.BUFF(1)
1390 NEXT C
1400 RETURN
```

THE FONT EDITOR PROGRAM

At times you may need a character file that doesn't correspond to the standard ASCII set. For example, you may want to define a character set made up of the Greek alphabet. For this you need to enter the bit patterns manually.

The following program is a "font editor." It allows you to create a file of blank characters and then set the pixels as you desire. It lets you define the dimensions of the characters as well as the foreground and background colors. You may then set or reset the points of the font. To make it easier on your eyes, the program displays copies of the font—one in its requested size and a second "blown up" copy.

The main routine initializes the data and graphics mode and requests the name of the character file. Then, until requested to do otherwise, it initializes the screen (line 1070), requests a character code (line 2650), and begins accepting changes to the font (lines 1090–1130). As you move about on the font, the program keeps you up-to-date on the row and column value of the cursor (line 1120).

```
1000 'Program font.editor
1010 '
1020 CLEAR ,,,32768!
1030 GOSUB 1240 'initialize.data.&.mode
1040 GOSUB 1470 'file.prompts
1050 ANSWER$ = "Y"
1060 WHILE ANSWER$<>"N" AND ANSWER$<>"n"
1070 GOSUB 1720 'initialize.grid
1080 GOSUB 2450 'initialize.character
1090 GOSUB 2760 'initialize.key.
                interrupts
1100 DONE = FALSE
1110 WHILE NOT DONE
1120 LOCATE 23,1 : PRINT "Column=";
     COLUMN+1;" Row=";ROW+1
1130 WEND
1140 VIEW
1150 CLS
1160 LOCATE 1,1
1170 PRINT TITLE$
1180 LOCATE 3,1
1190 INPUT "Define or edit another
     character (Y/N) ";ANSWER$
1200 WEND
1210 CLOSE 1
1220 END
1230 '
```


The next program defines two viewports: one for the life-size character display and a second for the blown-up version (lines 1280–1330). Throughout the program, the variable prefix POINT refers to the life-size character, while the prefix BLOCK refers to the blown-up character.

Two Boolean constants, TRUE and FALSE (lines 1340–1350), are defined as well as the main title (line 1360) and the colors used (lines 1380–1400).

```
1240 'subroutine initialize.data.&.mode
1250 '
1260 DEFINT A-Z
1270 DIM CHAR.ARRAY(200)
1280 BLOCK.X.BASE = 300
1290 BLOCK.Y.BASE = 16
1300 BLOCK.X.SIZE = 10
1310 BLOCK.Y.SIZE = 10
1320 POINT.X.BASE = 88
1330 POINT.Y.BASE = 56
1340 TRUE = (1=1)
1350 FALSE = (1=0)
1360 TITLE$ = STRING$(18,176)+" C h
     a r a c t e r   F o n t
     E d i t o r " + STRING$(18,176)
1370 SCREEN 6
1380 PALETTE
1390 PALETTE 1,3
1400 PALETTE 2,14
1410 CLS
1420 KEY OFF
1430 LOCATE 1,1
1440 PRINT TITLE$
1450 RETURN
1460 '
```

Setting Up Program Prompts

The next program gives you the option of creating a character file or editing an existing file (lines 1520–1540). If you don't supply a file extension, the program also supplies a .DAT extension (line 1570). If you ask to edit a file, the program first opens it as a sequential file in order to verify that it exists (lines 1580–1630). If the file doesn't exist, the program tells you and lets you try again (lines 1680–1700).

```
1470 'subroutine file.prompts
1480 '
1490 VALID = FALSE
1500 WHILE NOT VALID
1510 LOCATE 3,1
1520 INPUT "<C>reate a new character
     file or <E>dit an existing file ";
     ANSWER$
1530 IF ANSWER$="E" OR ANSWER$="e" THEN
     NEW.FILE=FALSE : OLD.FILE=TRUE :
     VALID=TRUE
1540 IF ANSWER$="C" OR ANSWER$="c" THEN
     NEW.FILE=TRUE : OLD.FILE=FALSE :
     VALID=TRUE
1550 WEND
1560 INPUT "File name: ",FILE.SPEC$
1570 IF INSTR(FILE.SPEC$,".")=0 THEN
     FILE.SPEC$ = FILE.SPEC$ + ".DAT"
1580 IF NEW.FILE THEN GOTO 1640
1590 'else
1600 ON ERROR GOTO 1680
1610 OPEN "I",1,FILE.SPEC$
1620 CLOSE
1630 ON ERROR GOTO 0
1640 OPEN "R",1,FILE.SPEC$,2
1650 FIELD 1,2 AS CHAR.PORTION$
1660 RETURN
1670 '
1680 'invalid.file.handler
1690 PRINT "I can't seem to find";
     FILE.SPEC$;" -- Please try again!!"
1700 RESUME 1560
1710 '
```

Setting Up the Program

The next subroutine clears the screen and prints up the character parameters. If the file is new, the program calls a routine to let you enter the character parameter values; if the file is old, the program calls a routine to read the data in from the file header (lines 1850–1870). If the file is new, the program then calls a routine to build the new file (line 1980).

For character editing, you will need some sort of a cursor to tell where you are. In this case, a box will be used for the cursor. The color of the cursor is selected in lines 1880–1910.


```
1720 'subroutine initialize.grid
1730 '
1740 CLS
1750 LOCATE 1,1
1760 PRINT TITLE$
1770 LOCATE 3,1
1780 KEY OFF
1790 PRINT "Font width:"
1800 PRINT "Font height:"
1810 PRINT "Foreground color:"
1820 PRINT "Background color:"
1830 PRINT "Ascii code:"
1840 PRINT "Character:"
1850 IF OLD.FILE THEN GOSUB 2010 :
     GOTO 1880 'input.stored.grid.
     parameters
1860 'else
1870 GOSUB 2130 'input.new.grid.
     parameters
1880 FRAME = 0
1890 WHILE FRAME=BACKGROUND OR
     FRAME=FOREGROUND
1900 FRAME = FRAME + 1
1910 WEND
1920 BLOCK.X.MAX = BLOCK.X.BASE +
     (X.PIXELS*BLOCK.X.SIZE)
1930 BLOCK.Y.MAX = BLOCK.Y.BASE +
     (Y.PIXELS*BLOCK.Y.SIZE)
1940 POINT.X.MAX = POINT.X.BASE +
     X.PIXELS
1950 POINT.Y.MAX = POINT.Y.BASE +
     Y.PIXELS
1960 CHAR.SIZE = INT((X.PIXELS+7)/8)
     * Y.PIXELS
1970 WINDOW SCREEN (0,0)-
     (X.PIXELS,Y.PIXELS)
1980 IF NEW.FILE THEN GOSUB 2250 'build.
     file.header
1990 RETURN
2000 '
```

If the file is old, the following program fetches the character parameters from the file header and then displays the values on the screen:

```
2010 'subroutine input.stored.grid.
     parameters
2020 '
```

```
2030 GET 1,1 : X.PIXELS =
     CVI(CHAR.PORTION$)
2040 GET 1,2 : Y.PIXELS =
     CVI(CHAR.PORTION$)
2050 GET 1,3 : FOREGROUND =
     CVI(CHAR.PORTION$)
2060 GET 1,4 : BACKGROUND =
     CVI(CHAR.PORTION$)
2070 LOCATE 3,13 : PRINT X.PIXELS;
2080 LOCATE 4,14 : PRINT Y.PIXELS;
2090 LOCATE 5,19 : PRINT FOREGROUND;
2100 LOCATE 6,19 : PRINT BACKGROUND;
2110 RETURN
2120 '
```

If the file is new, the following program prompts you for the size and color parameters:

```
2130 'subroutine input.new.grid.
     parameters
2140 '
2150 LOCATE 3,13 : INPUT "",X.PIXELS
2160 IF X.PIXELS > 24 OR X.PIXELS < 1
     THEN GOTO 2150
2170 LOCATE 4,14 : INPUT "",Y.PIXELS
2180 IF Y.PIXELS > 48 OR Y.PIXELS < 1
     THEN GOTO 2170
2190 LOCATE 5,19 : INPUT "",FOREGROUND
2200 IF FOREGROUND > 3 OR FOREGROUND < 0
     THEN GOTO 2190
2210 LOCATE 6,19 : INPUT "",BACKGROUND
2220 IF BACKGROUND > 3 OR BACKGROUND < 0
     THEN GOTO 2210
2230 RETURN
2240 '
```

If the file is new, the next program writes out the character parameters to the file header (lines 2260–2290). You then draw a "blank" character, using only the background color, and put its bit pattern into the remainder of the file (lines 2300–2410). After the file is built, you declare the tile "old" (line 2420).

```
2250 'subroutine build.file.header
2260 '
2270 LSET CHAR.PORTION$ = MKI$(X.PIXELS)
     : PUT 1,1
```



```
2280 LSET CHAR.PORTION$ = MKI$(Y.PIXELS)
     : PUT 1,2
2290 LSET CHAR.PORTION$ = MKI$(FOREGROUND)
     : PUT 1,3
2300 LSET CHAR.PORTION$ = MKI$(BACKGROUND)
     : PUT 1,4
2310 LOCATE 20,1 : PRINT "Initializing
     Ascii code";
2320 VIEW (POINT.X.BASE,POINT.Y.BASE)-
     (POINT.X.MAX,POINT.Y.MAX)
2330 LINE (0,0)-(X.PIXELS-1,Y.PIXELS-1),
     BACKGROUND,BF
2340 GET (0,0)-(X.PIXELS/2-1,Y.PIXELS-1),
     CHAR.ARRAY
2350 FOR RECORD=1 TO 128
2360 LOCATE 20,25,0 : PRINT USING "###";
     RECORD
2370 FOR PORTION=1 TO CHAR.SIZE
2380 LSET CHAR.PORTION$ =
     MKI$(CHAR.ARRAY(PORTION+1)) : PUT 1
2390 NEXT PORTION
2400 NEXT RECORD
2410 LOCATE 20,1,1 : PRINT SPACE$(50);
2420 NEW.FILE = FALSE : OLD.FILE = TRUE
2430 RETURN
2440 '
```

Initializing the Character

The next program prompts you for the code of the character that you want to define. It sets the character dimensions (elements 0 and 1 of CHAR.ARRAY) to the values of X.PIXELS and Y.PIXELS, respectively (lines 2500–2510), and then inputs the character's bit patterns from the data file. Once the character is in memory, the program puts the array (using PUT) and copies the bit pattern to the large-scale version (lines 2570–2660). Finally, the subroutine initializes the cursor location for the large-scale version (lines 2670–2730).

```
2450 'subroutine initialize.character
2460 '
2470 LOCATE 7,13 : INPUT "",CODE
2480 IF CODE < 1 OR CODE > 128 THEN
     GOTO 2470
2490 LOCATE 20,1 : PRINT "Recalling
     character";CODE;
```

```
2500 CHAR.ARRAY(0) = X.PIXELS
2510 CHAR.ARRAY(1) = Y.PIXELS
2520 BASE.RECORD = (CODE-
     1)*CHAR.SIZE + 4
2530 FOR OFFSET=1 TO CHAR.SIZE
2540 GET 1,BASE.RECORD+OFFSET
2550 CHAR.ARRAY(OFFSET+1) =
     CVI(CHAR.PORTION$)
2560 NEXT OFFSET
2570 VIEW (POINT.X.BASE,
     POINT.Y.BASE)-(POINT.X.MAX,
     POINT.Y.MAX)
2580 PUT (0,0),CHAR.ARRAY,PSET
2590 FOR COLUMN=0 TO X.PIXELS-1
2600 FOR ROW=0 TO Y.PIXELS-1
2610 POINT.COLOR = POINT(COLUMN,ROW)
2620 VIEW (BLOCK.X.BASE,BLOCK.Y.BASE)-
     (BLOCK.X.MAX,BLOCK.Y.MAX)
2630 LINE (COLUMN,ROW)-(COLUMN+.9,
     ROW+.9),POINT.COLOR,BF
2640 VIEW (POINT.X.BASE,POINT.Y.BASE)-
     (POINT.X.MAX,POINT.Y.MAX)
2650 NEXT ROW
2660 NEXT COLUMN
2670 OLD.COLUMN = 0
2680 OLD.ROW = 0
2690 COLUMN = 0
2700 ROW = 0
2710 VIEW (BLOCK.X.BASE,BLOCK.Y.BASE)-
     (BLOCK.X.MAX,BLOCK.Y.MAX)
2720 LOCATE 20,1 : PRINT SPACE$(50);
2730 GOSUB 3640 'update.displays
2740 RETURN
2750 '
```

Setting Up Function Key Interrupts

The next program uses the functions keys 1–4 to set points, reset points, save an edited character to disk, and to abort the edit. The arrow keys move the cursor around the font.

```
2760 'subroutine initialize.
     key.interrupts
2770 '
2780 ON KEY(1) GOSUB 2950 : KEY(1) ON :
     KEY 1,"Set"    'set.pixel
```



```
2790 ON KEY(2) GOSUB 3040 : KEY(2)
     ON : KEY 2,"Reset"  'reset.pixel
2800 ON KEY(3) GOSUB 3130 : KEY(3)
     ON :  KEY 3,"Save"    'save.character
2810 ON KEY(4) GOSUB 3250 : KEY(4)
     ON :  KEY 4,"Nosave" 'nosave.character
2820 ON KEY(11) GOSUB 3300 : KEY(11)
     ON 'up.arrow
2830 ON KEY(12) GOSUB 3550 : KEY(12)
     ON 'left.arrow
2840 ON KEY(13) GOSUB 3470 : KEY(13)
     ON 'right.arrow
2850 ON KEY(14) GOSUB 3990 : KEY(14)
     ON 'down.arrow
2860 KEY 5,""
2870 KEY 6,""
2880 KEY 7,""
2890 KEY 8,""
2900 KEY 9,""
2910 KEY 10,""
2920 KEY ON
2930 RETURN
2940 '
```

A point is set to the foreground color on the character using PSET at the current cursor location (line 2990). You can duplicate the change on the large-scale version by drawing a filled box at the current cursor position (line 3010). You can then update the display to redraw the cursor (line 3020).

```
2950 'subroutine set.point (function
     key 1)
2960 '
2970 VIEW (POINT.X.BASE,POINT.Y.BASE)-
     (POINT.X.MAX,POINT.Y.MAX)
2980 PSET (COLUMN,ROW),FOREGROUND
2990 VIEW (BLOCK.X.BASE,BLOCK.Y.BASE)-
     (BLOCK.X.MAX,BLOCK.Y.MAX)
3000 LINE (COLUMN,ROW)-(COLUMN+.9,
     ROW+.9),FOREGROUND,BF
3010 GOSUB 3640 'update.displays
3020 RETURN
3030 '
```

To reset the point, the program acts in essentially the same way as it did to set a point, this time using the background color rather than the foreground color.

```
3040 'subroutine reset.point (function
     key 2)
3050 '
3060 VIEW (POINT.X.BASE,POINT.Y.BASE)-
     (POINT.X.MAX,POINT.Y.MAX)
3070 PSET (COLUMN,ROW),BACKGROUND
3080 VIEW (BLOCK.X.BASE,BLOCK.Y.BASE)-
     (BLOCK.X.MAX,BLOCK.Y.MAX)
3090 LINE (COLUMN,ROW)-(COLUMN+.9,
     ROW+.9),BACKGROUND,BF
3100 GOSUB 3650 'update.displays
3110 RETURN
3120 '
```

To save the change, get the character and then write the pertinent data to the disk file. Also set the DONE flag so that you exit from the edit mode.

```
3130 'subroutine save.character
     (function key 3)
3140 '
3150 VIEW (POINT.X.BASE,POINT.Y.BASE)-
     (POINT.X.MAX,POINT.Y.MAX)
3160 GET (0,0)-(X.PIXELS/2-
     1,Y.PIXELS-1),CHAR.ARRAY
3170 BASE.RECORD = (CODE-
     1)*CHAR.SIZE + 4
3180 FOR OFFSET=1 TO CHAR.SIZE
3190 LSET CHAR.PORTION$ = MKI$(CHAR.
     ARRAY(OFFSET+1))
3200 PUT 1,BASE.RECORD+OFFSET
3210 NEXT OFFSET
3220 DONE = TRUE
3230 RETURN
3240 '
```

If you don't want to save the character, set the DONE flag to exit the edit mode.

```
3250 'subroutine nosave.character
3260 '
3270 DONE = TRUE
3280 RETURN
3290 '
```


Using the Arrow Keys

Pressing the up arrow key moves the cursor up one row. If it is already at the top of the character, it wraps around to the bottom.

```
3300 'subroutine up.arrow
3310 '
3320 OLD.ROW = ROW
3330 OLD.COLUMN = COLUMN
3340 ROW = ROW - 1
3350 IF ROW < 0 THEN ROW=Y.PIXELS-1
3360 GOSUB 3640
3370 RETURN
3380 '
```

Pressing the down arrow key moves the cursor down one row. If it is already at the bottom of the character, it wraps around to the top.

```
3390 'subroutine down.arrow
3400 '
3410 OLD.ROW = ROW
3420 OLD.COLUMN = COLUMN
3430 ROW = (ROW+1) MOD Y.PIXELS
3440 GOSUB 3640 'update.displays
3450 RETURN
3460 '
```

Pressing the right arrow key moves the cursor one column to the right. If it is already at the far right, it wraps around to the far left.

```
3470 'subroutine right.arrow
3480 '
3490 OLD.COLUMN = COLUMN
3500 OLD.ROW = ROW
3510 COLUMN = (COLUMN+1) MOD
     X.PIXELS
3520 GOSUB 3640 'update.displays
3530 RETURN
3540 '
```

Pressing the left arrow key moves the cursor one column to the left. If it is already at the far left, it wraps around to the far right.

```
3570 OLD.COLUMN = COLUMN
3580 OLD.ROW = ROW
3590 COLUMN = COLUMN - 1
3600 IF COLUMN < 0 THEN COLUMN =
     X.PIXELS - 1
3610 GOSUB 3640 'update.displays
3620 RETURN
3630 '
```

Each time the cursor moves or the current point changes color, the program redraws the current point on the large-scale version of the character and adds the cursor box around it.

```
3640 'subroutine update.displays
3650 '
3660 VIEW (POINT.X.BASE,POINT.Y.BASE)-
     (POINT.X.MAX,POINT.Y.MAX)
3670 OLD.COLOR = POINT(OLD.COLUMN,
     OLD.ROW)
3680 VIEW (BLOCK.X.BASE,BLOCK.Y.BASE)-
     (BLOCK.X.MAX,BLOCK.Y.MAX)
3690 LINE (OLD.COLUMN,OLD.ROW)-(OLD.
     COLUMN+.9,OLD.ROW+.9),OLD.COLOR,B
3700 LINE (COLUMN,ROW)-(COLUMN+.9,ROW
     +.9),FRAME,B
3710 RETURN
3720 '
3730 'End of program
```

CHAPTER NINETEEN

Icons and Menus

Most programs require you to select a feature from a number of choices. An accounting package, for example, may at one point ask you whether you want to run receivables, run payables, or generate a report.

When hardcopy terminals were the most advanced devices for data entry and display, running programs successfully required you to remember some key phrase or code to make your selection. When VDTs (video display terminals) came along, programs could display the possible selections in a list, or "menu," format, so that you could see which option you were choosing.

With the addition of graphics features to VDTs, a new way of selecting program options became available. Now you can draw a pictorial representation of a menu and move a cursor around the screen to the desired selection.

Graphic representations of menu selections are called *icons*. An icon could be a picture of:

- A printer, representing some print or report function
- An envelope, representing a mailing list function
- A memo, representing an electronic mail function.

There are at least two ways to use icons. The first way is to draw the menu icons on the screen and use a joystick or mouse to move a cursor to the portion of the screen where the desired icon is displayed. At that

point you can press a function routine and the program will select the function indicated by the current cursor coordinates. In order to use this method, your computer must of course include a joystick or a mouse.

A second method, for which we will present a sample program, uses a different method to highlight and select a desired icon. Each icon can be drawn in its own viewport with its own background palette color. By assigning all of the background palette colors to the same physical color, you can make all of the backgrounds look identical. To highlight a specific icon, just redefine its background color with a PALETTE statement.

CREATING MENUS WITH ICONS

The sample program will set up a menu for a fictitious program called "PCjr Graphics Utilities." This program has six menu options with an icon for each selection. Consequently, a screen mode with at least six colors (for the backgrounds) is required. Additional colors are required to draw the icons. We will use SCREEN 5, with its 16 colors and 320 × 200 resolution. (This is one of the occasions when having more colors is more important than having higher resolution.)

The program first initializes the screen mode and parameters, then draws each of the menu selections. Finally, it lets you select one of the menu options. After you make a selection, the program starts over (line 1110). Presumably, in a real application you would want to return to a main routine to call the indicated MENU.ITEM with an ON-GOSUB statement. The main routine includes the following programming lines:

```
1000 'Program menu.generator
1010 '
1020  CLEAR ,,,32768!
1030 GOSUB 1130          'initialize
1040 GOSUB 1360          'line.graph
1050 GOSUB 1570          'bar.graph
1060 GOSUB 1690          'pie.graph
1070 GOSUB 1840          'function.plot
1080 GOSUB 2010          'filed.graph
1090 GOSUB 2260          'exit.utility
1100 GOSUB 2410          'select.item
1110 GOTO 1030
1120 '
```


Setting Up the Program

The standard initializations are present in the next routine. In addition, palette colors 9–14 are assigned to the color yellow. These palette colors will be the backgrounds.

```
1130 'subroutine initialize
1140 '
1150      CR$=CHR$(&HD)
1160      SCREEN 5
1170      VIEW
1180      WINDOW
1190      CLS
1200      KEY OFF
1210      PALETTE
1220      COLOR 1,3
1230      PALETTE 6,6
1240      PALETTE 7,0
1250      PALETTE 9,14
1260      PALETTE 10,14
1270      PALETTE 11,14
1280      PALETTE 12,14
1290      PALETTE 13,14
1300      PALETTE 14,14
1310      PALETTE 15,1
1320      LOCATE 1,9
1330      PRINT "PCjr Graphics Utilities"
1340 RETURN
1350 '
```

Creating the Icons

The next routine provides the first menu selection, a line graph utility:

```
1360 'subroutine line.graph
1370 '
1380      VIEW (5,14)-(105,93),9,15
1390      WINDOW (0,0)-(10,10)
1400      FOR ROW=2 TO 8 STEP 2
1410          LINE (0,ROW)-(10,ROW),15
1420          LINE (ROW,0)-(ROW,10),15
1430      NEXT ROW
1440      LINE (0,0)-(4,3),7
1450      LINE -(6,2),7
1460      LINE -(10,4),7
```

```
1470      LINE (0,1)-(4,7),4
1480       LINE -(6,5),4
1490       LINE -(10,7),4
1500       LINE (0,3)-(4,4),5
1510       LINE -(6,9),5
1520       LINE -(10,4),5
1530      LOCATE 13,3
1540      PRINT "Line Graph";
1550 RETURN
1560 '
```

The next subroutine presents a bar graph utility:

```
1570 'subroutine bar.graph
1580 '
1590       VIEW (110,14)-(210,93),10,15
1600      WINDOW (0,0)-(10,10)
1610      FOR BAR=0 TO 9
1620          LINE (BAR+.1,0)-(BAR + .4,
              RND * BAR + 3),4,BF
1630          LINE (BAR+.6,0)-(BAR + .9,
              RND * BAR + 3),2,BF
1640      NEXT BAR
1650      LOCATE 13,16
1660      PRINT "Bar Graph";
1670 RETURN
1680 '
```

The third option, a pie graph utility, is presented in the next routine:

```
1690 'subroutine pie.graph
1700 '
1710       VIEW (215,14)-(315,93),11,15
1720      WINDOW (-1.5,-1.5)-(1.5,1.5)
1730       CIRCLE (0,0),1,15
1740      LINE (0,0)-(1,0),15
1750       LINE (0,0)-(0,1),15
1760      LINE (0,0)-(-.7,-.7),15
1770       PAINT (.5,.5),4,15
1780      PAINT (-.5,0),0,15
1790       PAINT (.5,-.5),2,15
1800      LOCATE 13,30
1810      PRINT  "Pie Graph";
1820 RETURN
1830 '
```


The function plot utility, the fourth option, is programmed in the next subroutine:

```
1840 'subroutine function.plot
1850 '
1860       VIEW (5,110)-(105,189),12,15
1870      WINDOW (-4,-4)-(4,4)
1880       LINE (0,4)-(0,-4),15
1890      LINE (-4,0)-(4,0),15
1900      FOR TICK=-3 TO  3
1910      LINE (-.2,TICK)-(.2,TICK),15
1920       LINE (TICK,-.2)-(TICK,.2),15
1930      NEXT TICK
1940       FOR X=-3.14 TO 3.14 STEP .1
1950           PSET (X,2*COS(X)),4
1960      NEXT X
1970      LOCATE 25,1
1980      PRINT "Function Plot";
1990 RETURN
2000 '
```

The next subroutine displays the fifth option, a filed graph recall utility:

```
2010 'subroutine filed.graph
2020 '
2030       VIEW (110,110)-(210,189),13,15
2040      WINDOW (1,1)-(9,10)
2050       LINE (7,2)-(2,2),15
2060      LINE -(2,8),15
2070      LINE -(3,9),15
2080      LINE -(8,9),15
2090      LINE -(8,3),15
2100      LINE -(7,2),15
2110      LINE -(7,8),15
2120      PAINT (7.1,8),0,15
2130       LINE (7,8)-(8,9),15
2140      LINE (2,8)-(7,8),15
2150       LINE (2,5)-(7,5),15
2160      LINE (3,6.5)-(2.5,6),15
2170      LINE -(6,6),15
2180      LINE -(6.5,6.5),15
2190       LINE (3,4)-(2.5,3.5),15
2200      LINE -(6,3.5),15
2210       LINE -(6.5,4),15
2220      LOCATE 25,16
```

```
2230      PRINT "Filed Graph";
2240 RETURN
2250 '
```

The last subroutine displays the final option, exiting the utility:

```
2260 'subroutine exit.utility
2270 '
2280      VIEW (215,110)-(315,189),14,15
2290      WINDOW (0,0)-(10,10)
2300      LINE (3,8)-(7,7),7,B
2310     LINE (3,6)-(7,5),7,B
2320      LINE (2,9)-(8,0),7,B
2330      PAINT (4,1),6,7
2340      LINE (1.75,2.5)-(2.25,2),7,BF
2350     LINE (1.75,7)-(2.25,6.5),7,BF
2360      CIRCLE (7.5,4.5),.5,7
2370      PAINT (7.5,4.5),7,7
2380      LOCATE 25,32
2390      PRINT "Exit";
2400 RETURN
2410 '
```

Selecting a Menu Option

The next routine highlights a menu selection simply by changing its background color to white. To move to the next menu item, press any key except ENTER, and the program responds by returning the current option background to yellow and making the next option's background white. When you press ENTER, the subroutine returns to the main line code.

```
2420 'subroutine select.item
2430 '
2440      MENU.ITEM = 0
2450      PALETTE MENU.ITEM+9,15
2460      A$ = INKEY$
2470      WHILE A$ <> CR$
2480      IF A$="" THEN GOTO 2520
2490               'else
2500      PALETTE MENU.ITEM+9,14
2510      MENU.ITEM=(MENU.ITEM+1) MOD 6
2520           A$=INKEY$
```



```
2530      WEND
2540 RETURN
2550 '
2560 'End of program
```

Figure 19-1 shows a sample display of this program.

CHAPTER TWENTY

Animation

As Chapter 10 demonstrated, effective animation with BASICA is often difficult, if not impossible. This is due to the relative slowness of a BASIC program compared to a machine language program. If you keep motions simple, however, you can write a program with limited animation.

To create the illusion of motion, an object must be drawn at one location, be moved slightly, and then be drawn again. You must then erase the object from the original location. If you continue these steps, the object will appear to move across the screen.

Action is one of the PUT statement's options. It refers to the logical result of using PUT to position a bit pattern on top of the points already on the screen. Depending on the action, the array will superimpose the points on the screen, replace the points on the screen, or mix with the points on the screen.

The default action for PUT is the Boolean operator XOR (eXclusive OR). The result of two binary values XORed together is as follows:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

It may not be apparent, but this property of XOR can help you with animation. After one PUT statement, an object appears on the screen. If you then use PUT to position the same array at the same coordinates, you will erase the object. In addition, XOR will preserve the condition of the screen after two consecutive PUT instructions. For example, if you position a "rocket ship" on top of a "star," the star may change colors, but it will still be there. After you put the rocket ship again, the rocket image is gone, and the star is in its original condition.

SIMPLE ANIMATION PROGRAM

The following sample program draws a rocket ship that moves up through some stars, then back down. Finally it explodes. The main routine includes the following programming lines:

```
1000 CLEAR ,,,32768!
1010 GOSUB 1100  'initialize
1020 GOSUB 1370  'up
1030 GOSUB 1470  'up.and.right
1040 GOSUB 1590  'right
1050 GOSUB 1690  'down.and.right
1060 GOSUB 1810  'down
1070 GOSUB 1910  'explode
1080 END
1090 '
```

Setting Up the Program

The next subroutine initializes the graphics mode, draws the rocket shapes (line 1190), changes the orientation, and gets them into arrays R1, R2, R3, R4, and R5 (lines 1200–1290). It also prints "stars" (asterisks) in a random pattern on the screen (lines 1310–1340).

```
1100 'initialize
1110 '
1120 DIM R1(100),R2(100),R3(100),
     R4(100),R5(100)
```



```
1130 SCREEN 5
1140 CLS
1150 KEY OFF
1160 PALETTE
1170 PALETTE 1,6
1180 PALETTE 2,4
1190 ROCKET$ = "c1 u10 e5 f5 d10 l10"
1200 DRAW "bm0,50" + ROCKET$
1210 GET (0,50)-(10,35),R1
1220 DRAW "bm20,50 ta-45" + ROCKET$
1230 GET (20,60)-(36,40),R2
1240 DRAW "bm50,50 ta-90" + ROCKET$
1250 GET (50,50)-(70,60),R3
1260 DRAW "bm80,50 ta-135" + ROCKET$
1270 GET (70,50)-(90,75),R4
1280 DRAW "bm110,50 ta-180" + ROCKET$
1290 GET (110,50)-(100,65),R5
1300 CLS
1310 FOR I=1 TO 100
1320 LOCATE RND(1)*23+1, RND(1)*39+1
1330 PRINT "*";
1340 NEXT I
1350 RETURN
1360 '
```

Moving the Rocket

The following subroutine moves the rocket up. Note that immediately before the FOR loop, the program puts the array. This provides an initial rocket to erase in the loop. The array is also put immediately after the loop. This erases the last rocket drawn within the loop.

```
1370 'subroutine up
1380 '
1390 PUT (100,181),R1
1400 FOR Y=180 TO 100 STEP -1
1410 PUT (100,Y),R1
1420 PUT (100,Y+1),R1
1430 NEXT Y
1440 PUT (100,100),R1
1450 RETURN
1460 '
```

The following subroutine moves the rocket diagonally up and to the right:

```
1470 'subroutine up.and.right
1480 '
1490 PUT (100,101),R2
1500 X=101
1510 FOR Y=100 TO 80 STEP -1
1520 PUT (X,Y),R2
1530 PUT (X-1,Y+1),R2
1540 X=X+1
1550 NEXT Y
1560 PUT (121,80),R2
1570 RETURN
1580 '
```

To move the rocket across the screen, enter the following subroutine:

```
1590 'subroutine across
1600 '
1610 PUT (124,80),R3
1620 FOR X=125 TO 150
1630 PUT (X,80),R3
1640 PUT (X-1,80),R3
1650 NEXT X
1660 PUT (150,80),R3
1670 RETURN
1680 '
```

The next subroutine moves the rocket diagonally down and to the right:

```
1690 'subroutine down.and.right
1700 '
1710 PUT (150,80),R4
1720 X=151
1730 FOR Y=81 TO 100
1740 PUT (X,Y),R4
1750 PUT (X-1,Y-1),R4
1760 X=X+1
1770 NEXT Y
1780 PUT (170,100),R4
1790 RETURN
1800 '
```


Finally, this subroutine moves the rocket down:

```
1810 'subroutine down
1820 '
1830 PUT (180,100),R5
1840 FOR Y=101 TO 180
1850 PUT (180,Y),R5
1860 PUT (180,Y-1),R5
1870 NEXT Y
1880 PUT (180,180),R5
1890 RETURN
1900 '
```

Exploding the Rocket

This subroutine explodes the rocket. It does so by drawing explosion lines in palette color 2, away from the last point the rocket was seen. It then flashes the lines by changing the palette definition of color 2 from red to black in rapid succession.

```
1910 'subroutine explode
1920 '
1930 LINE (180,199)-(150,150),2
1940 LINE (180,199)-(170,140),2
1950 LINE (180,199)-(180,130),2
1960 LINE (180,199)-(190,140),2
1970 LINE (180,199)-(210,150),2
1980 FOR FLASH=0 TO 20
1990 PALETTE 2,4
2000 FOR I=0 TO 10 : NEXT I
2010 PALETTE 2,0
2020 FOR I=0 TO 10 : NEXT I
2030 NEXT FLASH
2040 RETURN
2050 '
2060 'End of program
```

PART FOUR

Appendixes

The appendixes in this section provide a number base conversion chart (binary-hexadecimal-decimal) that you will find useful when tiling (with PAINT) and creating different line styles (with LINE). Also included is a graphics worksheet (medium resolution) to make plotting screen images easier.

APPENDIX A

Binary/Decimal/ Hexadecimal Base Conversion Chart

The following chart is useful for determining colors and patterns of PAINT tiles and LINE styles:

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000 0000	11	B	0000 1011
1	1	0000 0001	12	C	0000 1100
2	2	0000 0010	13	D	0000 1101
3	3	0000 0011	14	E	0000 1110
4	4	0000 0100	15	F	0000 1111
5	5	0000 0101	16	10	0001 0000
6	6	0000 0110	17	11	0001 0001
7	7	0000 0111	18	12	0001 0010
8	8	0000 1000	19	13	0001 0011
9	9	0000 1001	20	14	0001 0100
10	A	0000 1010	21	15	0001 0101

Decimal	Hex	Binary	Decimal	Hex	Binary
22	16	0001 0110	50	32	0011 0010
23	17	0001 0111	51	33	0011 0011
24	18	0001 1000	52	34	0011 0100
25	19	0001 1001	53	35	0011 0101
26	1A	0001 1010	54	36	0011 0110
27	1B	0001 1011	55	37	0011 0111
28	1C	0001 1100	56	38	0011 1000
29	1D	0001 1101	57	39	0011 1001
30	1E	0001 1110	58	3A	0011 1010
31	1F	0001 1111	59	3B	0011 1011
32	20	0010 0000	60	3C	0011 1100
33	21	0010 0001	61	3D	0011 1101
34	22	0010 0010	62	3E	0011 1110
35	23	0010 0011	63	3F	0011 1111
36	24	0010 0100	64	40	0100 0000
37	25	0010 0101	65	41	0100 0001
38	26	0010 0110	66	42	0100 0010
39	27	0010 0111	67	43	0100 0011
40	28	0010 1000	68	44	0100 0100
41	29	0010 1001	69	45	0100 0101
42	2A	0010 1010	70	46	0100 0110
43	2B	0010 1011	71	47	0100 0111
44	2C	0010 1100	72	48	0100 1000
45	2D	0010 1101	73	49	0100 1001
46	2E	0010 1110	74	4A	0100 1010
47	2F	0010 1111	75	4B	0100 1011
48	30	0011 0000	76	4C	0100 1100
49	31	0011 0001	77	4D	0100 1101

Decimal	Hex	Binary	Decimal	Hex	Binary
78	4E	0100 1110	106	6A	0110 1010
79	4F	0100 1111	107	6B	0110 1011
80	50	0101 0000	108	6C	0110 1100
81	51	0101 0001	109	6D	0110 1101
82	52	0101 0010	110	6E	0110 1110
83	53	0101 0011	111	6F	0110 1111
84	54	0101 0100	112	70	0111 0000
85	55	0101 0101	113	71	0111 0001
86	56	0101 0110	114	72	0111 0010
87	57	0101 0111	115	73	0111 0011
88	58	0101 1000	116	74	0111 0100
89	59	0101 1001	117	75	0111 0101
90	5A	0101 1010	118	76	0111 0110
91	5B	0101 1011	119	77	0111 0111
92	5C	0101 1100	120	78	0111 1000
93	5D	0101 1101	121	79	0111 1001
94	5E	0101 1110	122	7A	0111 1010
95	5F	0101 1111	123	7B	0111 1011
96	60	0110 0000	124	7C	0111 1100
97	61	0110 0001	125	7D	0111 1101
98	62	0110 0010	126	7E	0111 1110
99	63	0110 0011	127	7F	0111 1111
100	64	0110 0100	128	80	1000 0000
101	65	0110 0101	129	81	1000 0001
102	66	0110 0110	130	82	1000 0010
103	67	0110 0111	131	83	1000 0011
104	68	0110 1000	132	84	1000 0100
105	69	0110 1001	133	85	1000 0101

Decimal	Hex	Binary
134	86	1000 0110
135	87	1000 0111
136	88	1000 1000
137	89	1000 1001
138	8A	1000 1010
139	8B	1000 1011
140	8C	1000 1100
141	8D	1000 1101
142	8E	1000 1110
143	8F	1000 1111
144	90	1001 0000
145	91	1001 0001
146	92	1001 0010
147	93	1001 0011
148	94	1001 0100
149	95	1001 0101
150	96	1001 0110
151	97	1001 0111
152	98	1001 1000
153	99	1001 1001
154	9A	1001 1010
155	9B	1001 1011
156	9C	1001 1100
157	9D	1001 1101
158	9E	1001 1110
159	9F	1001 1111
160	A0	1010 0000
161	A1	1010 0001
162	A2	1010 0010
163	A3	1010 0011
164	A4	1010 0100
165	A5	1010 0101
166	A6	1010 0110
167	A7	1010 0111
168	A8	1010 1000
169	A9	1010 1001
170	AA	1010 1010
171	AB	1010 1011
172	AC	1010 1100
173	AD	1010 1101
174	AE	1010 1110
175	AF	1010 1111
176	B0	1011 0000
177	B1	1011 0001
178	B2	1011 0010
179	B3	1011 0011
180	B4	1011 0100
181	B5	1011 0101
182	B6	1011 0110
183	B7	1011 0111
184	B8	1011 1000
185	B9	1011 1001
186	BA	1011 1010
187	BB	1011 1011
188	BC	1011 1100
189	BD	1011 1101

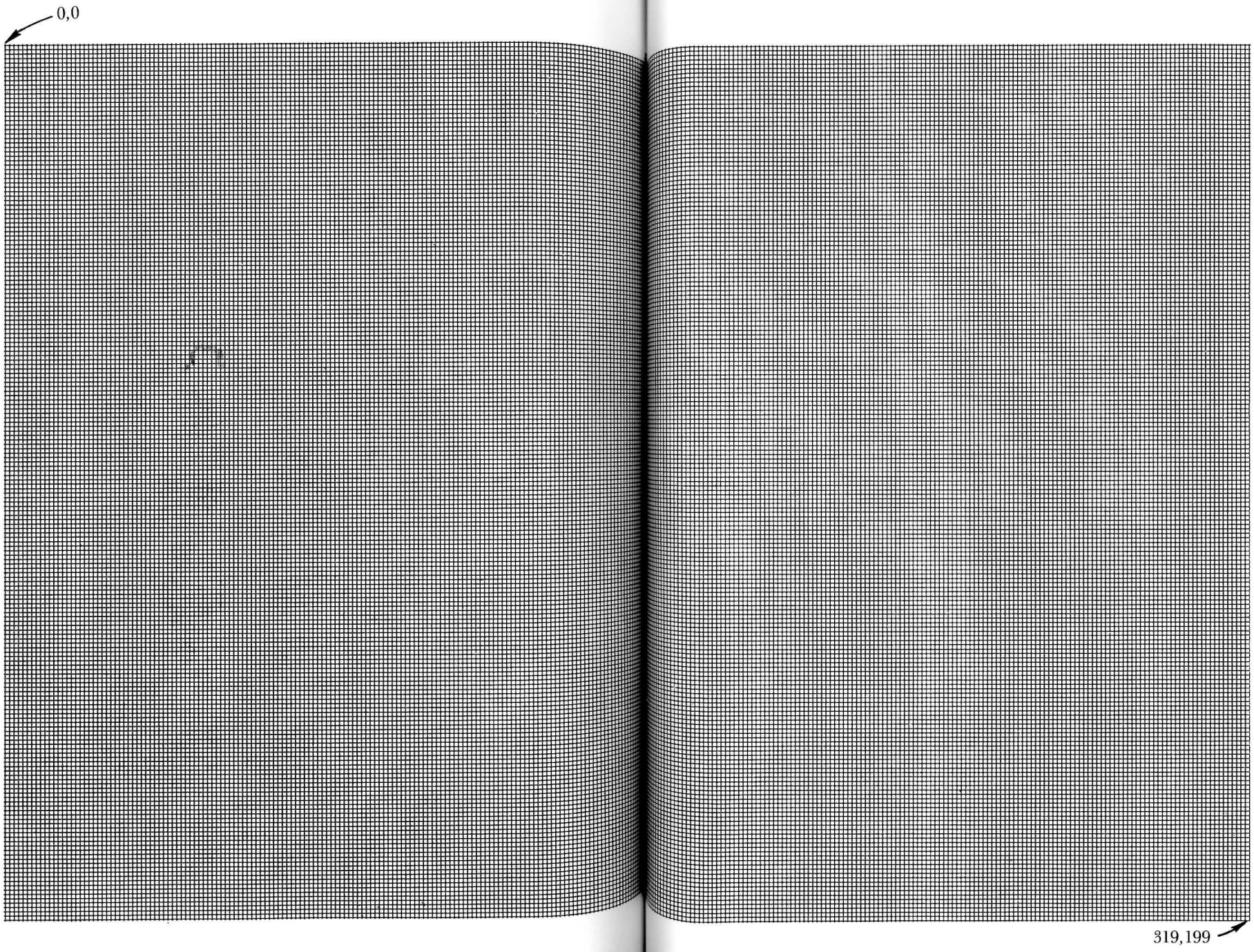
Decimal	Hex	Binary
190	BE	1011 1110
191	BF	1011 1111
192	C0	1100 0000
193	C1	1100 0001
194	C2	1100 0010
195	C3	1100 0011
196	C4	1100 0100
197	C5	1100 0101
198	C6	1100 0110
199	C7	1100 0111
200	C8	1100 1000
201	C9	1100 1001
202	CA	1100 1010
203	CB	1100 1011
204	CC	1100 1100
205	CD	1100 1101
206	CE	1100 1110
207	CF	1100 1111
208	D0	1101 0000
209	D1	1101 0001
210	D2	1101 0010
211	D3	1101 0011
212	D4	1101 0100
213	D5	1101 0101
214	D6	1101 0110
215	D7	1101 0111
216	D8	1101 1000
217	D9	1101 1001
218	DA	1101 1010
219	DB	1101 1011
220	DC	1101 1100
221	DD	1101 1101
222	DE	1101 1110
223	DF	1101 1111
224	E0	1110 0000
225	E1	1110 0001
226	E2	1110 0010
227	E3	1110 0011
228	E4	1110 0100
229	E5	1110 0101
230	E6	1110 0110
231	E7	1110 0111
232	E8	1110 1000
233	E9	1110 1001
234	EA	1110 1010
235	EB	1110 1011
236	EC	1110 1100
237	ED	1110 1101
238	EE	1110 1110
239	EF	1110 1111
240	F0	1111 0000
241	F1	1111 0001
242	F2	1111 0010
243	F3	1111 0011
244	F4	1111 0100
245	F5	1111 0101

Decimal	Hex	Binary
246	F6	1111 0110
247	F7	1111 0111
248	F8	1111 1000
249	F9	1111 1001
250	FA	1111 1010
251	FB	1111 1011
252	FC	1111 1100
253	FD	1111 1101
254	FE	1111 1110
255	FF	1111 1111

APPENDIX B

Graphics Worksheet (Medium Resolution)

This worksheet is useful for creating graphics designs in medium resolution (SCREENs 1, 4, and 5). Two medium-resolution X-axis squares on this grid will equal one square in low resolution and one X-axis square on this grid will equal two squares in high resolution. We suggest that you make working copies of this page rather than writing directly in the book.



Graphics Worksheet (Medium Resolution)

Index

A

Animation, 3, 69, 291-95
Arrays, 51, 151, 153, 155-64
Aspect ratio, 94, 103-07, 132-33

B

B option (DRAW), 115
B option (LINE), 77-79, 135, 153, 173, 200
Background colors, 7, 14, 19, 34, 38, 42-45
Bar graphs, 197-214
BASICA, 7, 56, 162, 179, 233, 266
Binary, 80, 141, 299-304

C

CAD (Computer-aided design), 3
Cartesian, 234, 251
CHAIN, 252-53
CHR$, 142
CIRCLE, 26, 29, 65, 89, 138, 234-35
CLEAR, 23, 27, 31, 33, 37, 168
CLS, 23, 30, 37
COLOR, 8, 18, 37, 38-46, 73, 124, 139
Colors, 5, 16
Color codes, 17-18, 40, 42, 48, 51, 76, 82, 143, 176
COS, 234, 238

D
DATA, 51, 86
DEF, 157, 252
DIM, 51, 153, 156, 159
DRAW, 111-33, 265

F
FOR-NEXT, 62, 70
Foreground colors, 7, 14, 38, 39-42, 56, 75

G
GET, 151, 152-53, 198, 265
GOTO, 62
Graphics, 3, 4, 7
Graphics definition language (GDL), 111
Graphics Mode, 7, 8, 15, 23, 25, 36, 38, 75, 95, 224
Graphics Primer for the IBM® PC, 102, 108, 143

H
Hands-On BASIC, ix, 51, 151
Hexadecimal, 80-82, 142, 299-304

I
Icons, 283-89

K
KILL, 253

L
LINE, 18, 37, 69, 70-87, 96
Line clipping, 85-86
Line graphs, 223-32
Line styles, 79, 141, 199, 299
LOCATE, 199, 200
LRP, 14, 56, 62, 65, 67, 69, 72, 85, 111, 113, 117, 190

M
Media, 3

N
N option (DRAW), 117

O
OR/AND, 165-66

P
Page flipping, 5, 167, 169-70
Pages, active/visual, 29, 31-32
PAINT, 18, 29, 37, 96, 135-150, 236
PALETTE, 19, 37, 38, 45, 47-50, 56, 76, 79, 124, 140, 284
PALETTE USING, 45, 50-54, 140
PCOPY, 167-70
Pi, 97, 98, 234, 244
Pie charts, 233-49
Pixel, 5, 10, 11, 55, 57, 58, 60, 63, 69, 80, 94, 127
PMAP, 189, 192-94, 199, 200
POINT, 189, 190-92, 261, 265
Polar coordinates, 257
PRESET, 55, 65-68
PRINT, 199, 261
PSET, 55, 56-65, 89, 102, 139, 279
PSET/PRESET (PUT), 165
Punctuation, 121
PUT, 151, 154-55, 198, 265, 291

R
Radians, 97
RANDOMIZE, 200
Relative positioning, 14
Resolution (low/medium/high), 7, 8, 16, 28, 32, 57, 103
RGB monitor, 28, 233
RND, 200

S
Scaling, 127
SCREEN, 7, 16, 23-35, 37, 40, 41, 45, 48, 56, 60, 75, 94, 119, 157, 168
SIN, 234, 238
STEP (CIRCLE), 107, 120
STEP (LINE), 82, 120
STEP (PRESET), 67
STEP (PSET), 62, 120

T
Text Mode, 7, 8, 15, 25, 36, 38
Three-dimensional, 122, 211, 243
Tiling, 5, 135, 141-50, 236, 299

V
Video memory, 27, 31, 32, 34-35
VIEW, 14, 171, 172-79, 189, 198
Viewing transformation, 172

W
WIDTH, 8, 261
WINDOW, 14, 171, 179-87, 189, 198, 235, 256
World coordinates, 180-82

X
X,Y coordinates, 11-14, 36, 56, 58, 60, 65, 70, 82, 89, 92, 107, 113, 137, 171, 179, 189, 226, 251
XOR, 161, 163-64, 291

About the Authors

Jonathan Erickson, a senior editor at Osborne/McGraw-Hill, is a former newspaper reporter and technical writer. He is also the author of *C-64 Telecommunications* and co-author of *The Model 100 Book: A Guide to Portable Computing*. He has written numerous magazine articles on a variety of topics and also contributed to the *McGraw-Hill Computer Handbook*.

William Cramer, who graduated from Texas Christian University with degrees in computer science and chemistry, is a software engineer specializing in graphics, communications and real-time vehicle monitoring.

Look for these Osborne/McGraw-Hill IBM computer books:

Your IBM® PC: A Guide to the IBM PC (DOS 2.0) and XT
Your IBM® PC: A Guide to the IBM Personal Computer
Graphics Primer for the IBM® PC
Your IBM® Made Easy [Includes IBM PC (DOS 2.0) and PC-XT]
The Osborne/McGraw-Hill Guide to IBM® PC Communications
The 8086 Book

Available at computer stores and bookstores everywhere.
Or order direct by calling TOLL-FREE: 800-227-2895.
In California call 800-772-4077.

The IBM® PCjr IMAGE MAKER
Graphics on the IBM® PCjr

This excellent, colorful teaching guide shows IBM® PCjr users how to create dynamic color graphics on the PCjr.

You'll learn to use the PCjr's unique graphics features to communicate ideas more effectively in business, education, or home applications.

Clearer and more comprehensive than the PCjr documentation, this exhaustive hands-on resource book offers detailed explanations of BASICA's graphics programming statements.

Practical application programs are presented with photographs of actual graphics displays: pie charts, bar and line graphs, animation, and more.

Other topics include three-dimensional drawings, pattern repetition or "tiling," and basic animation.

***The IBM® PCjr Image Maker** contains all the programming tools you need to design sophisticated color graphics of your own.*

IBM is a registered trademark of IBM.